UNIVERSITA' DEGLI STUDI DI ROMA
TorVergata
Università

Laurea Specialistica in Informatica

Corso di Algebra
a.a. 2007-2008

Basi di Crittografia

dal modello matematico a quello informatico

Giuseppe Specchio

Prefazione

Il presente documento vuole rappresentare un'opera implementativa di alcuni algoritmi crittografici studiati durante il corso di Algebra[1], nonché un effettivo riscontro tra i modelli matematici approfonditi ed il loro concreto utilizzo nel contesto informatico.

Gli applicativi proposti, sono stati realizzati in linguaggio di programmazione Java secondo il pattern MVC[2], in modo tale da poterne sfruttare il loro massimo riuso in altri contesti. A tal proposito, si è deciso di inserire la mera parte algoritmica in tre package distinti, tutti caratterizzati da classi con metodi statici denominati rispettivamente:

- *algebra:* contenente le implementazioni delle principali strutture algebriche necessarie per l'implementazione di sistemi crittografici e la generazione di numeri primi *su n* bit. Tali metodologie sono state descritte ed implementate nel *capitolo 1* del presente documento.
- *matrici* : naturale estensione del contenuto delle classi contenuti nel package precedente. Mediante il suo uso sarà possibile risolvere sistemi lineari di congruenze.
- *cifrari*: contenente le implementazioni dei sistemi crittografici proposti nel documento, la cui esecuzione è consentita solo grazie all'utilizzo dei metodi contenuti nelle classi dei package precedenti. Tali metodologie sono state descritte ed implementate nel capitolo 2 del presente documento.

Per ogni metodo che caratterizza le classi citate, sono state realizzate delle classi di testing ad interfaccia grafica, in modo tale da poter rendere il prodotto finale più comprensibile anche verso chi si avvicina per la prima volta al mondo della crittografia.

In conclusione, si evidenzia il fatto che dovendo gestire una grande quantità numerica di dati si è scelto di far uso della classe wrapper *BigInteger* al fine di fronteggiare potenziali problemi di overflow.

---

1 Corso di Algebra (Metodi algebrici per la crittografia) tenuto durante l'a.a.2007-2008 del Corso di Laurea Specialistica in Informatica da parte dalla Prof.ssa M.G. Piacentini Cattaneo.

2 Model View Control è un pattern è basato sulla separazione dei compiti fra i componenti software che interpretano tre ruoli principali:

* il model contiene i dati e fornisce i metodi per accedervi;
* il view visualizza i dati contenuti nel model;
* il controller riceve i comandi dell'utente (in genere attraverso il view) e li attua modificando lo stato degli altri due componenti.

Questo schema, fra l'altro, implica anche la tradizionale separazione fra la logica applicativa (in questo contesto spesso chiamata "logica di business"), a carico del model, e l'interfaccia utente, a carico del view e del controller.

Indice

Prefazione....2
Introduzione....5
CAPITOLO 1
Il modello matematico....6
Introduzione....6
L'Algoritmo di Euclide....6
L'identità di Bézout....8
Equazioni lineari diofantee....11
Congruenze....14
Congruenze lineari....15
Sistemi di Congruenze lineari....18
Teorema Cinese dei resti....35
Congruenze polinomiali non lineari....38
Equazioni di secondo grado....39
Metodi di Fattorizzazione....44
Il Crivello di Eratostene....44
Il metodo di fattorizzazione di Fermat....46
Il metodo di fattorizzazione ρ di Pollard....53
Il metodo di fattorizzazione p-1 di Pollard....57
Calcolo delle potenze modulo n....57
Piccolo Teorema di Fermat....60
Generatori....66
Calcolo del Logaritmo Discreto....72
Algoritmo Baby step – Giant step....72
CAPITOLO 2
Il modello Informatico....77
Introduzione....77
Cos'è la crittografia....77
Impostazione matematica di un crittosistema....80
La crittografa a chiave segreta o simmetrica....82
Cifrari basati su aritmetica modulare....83
Cifrari affini....83
I principali algoritmi di crittografia a chiave segreta....93
La crittografia a chiave pubblica o asimmetrica....93
La riservatezza dei messaggi....94
L'autenticità del mittente....95
L'autenticità del mittente e l'integrità del messaggio....95
L'autenticità del mittente, la riservatezza e l'integrità del messaggio....96
Cifrari a chiave pubblica....97
Il sistema RSA....98
Invio di un messaggio cifrato con il sistema RSA....99
Decifratura di un messaggio cifrato con il sistema RSA....102
Perché ha funzionato?....105
Una miglioria dell'algoritmo....105
Autenticazione delle firme con il sistema RSA....113
L'implementazione....116
I principali algoritmi di crittografia a chiave pubblica....122
Man In The Middle....123

I principali algoritmi per funzioni di hashing....123
La firma digitale ed i certificati a chiave pubblica....124
Creazione di un certificato in Java....127
Le Public Key Infrastructure....132
La crittografia in Internet....134
I protocolli sicuri per Internet....135
I protocolli sicuri per la posta elettronica....136
I protocolli sicuri per il Web....136
SSL (Secure Socket layer)....136
SHTTP (Securee HTTP)....138
I sistemi di pagamento elettronico....138
Sicurezza e tecnologia wireless....138
BIBLIOGRAFIA....140

Introduzione

Le tecnologie telematiche oramai, volenti o nolenti, sono entrate a pieno a far parte della nostra quotidianità, ad esempio quando facciamo delle compere e paghiamo con il bancomat o carta di credito, quando carichiamo la scheda carburanti, quando attraversiamo il casello autostradale, quando comunichiamo al cellulare, quando paghiamo le tasse, quando ci viene prestato dell'assistenza sanitaria e magari un giorno, quando voteremo elettronicamente!!

L'uso di tali strumenti, enormemente potenti e che tanto ci hanno facilitato la vita rispetto al passato hanno però un grande lato negativo, ovvero l'intercettazione degli stessi!!

Proviamo solo per un attimo ad immaginare cosa possa succedere se un terzo individuo possa intercettare, senza non molta fatica, il nostro codice PIN del Bancomat o del telefono cellulare, o le stesse nostre comunicazioni o peggio ancora i nostri dati sanitari!

Tutti gli stati c.d. evoluti, hanno fissato sia a livello comunitario che locale delle norme che regolarizzino la gestione e quindi la messa in sicurezza di tali informazioni dette anche *dati sensibili*, in particolare in Italia chiunque realizzi o gestisca una struttura telematica che elabori dati od informazioni risalenti a persone fisica o giuridiche deve rispettare quanto asserito dagli *artt.33 e 34 lett.g* del *Decreto Legislativo 30 giugno 2003, n. 196* "*Codice in materia di protezione dei dati personali*", il quale dispone la stesura, con cadenza almeno annuale, del *Documento Programmatico sulla Sicurezza (D.P.S.)*[3]. Da quest'ultimi vengono dedotti i criteri e le certificazioni della “messa in sicurezza” di tali dati. Ma cosa intendiamo per “*messa in sicurezza*” dei dati? Dal punto di vista, meramente informatico, si intende anche e non solo l'attuazione di metodi crittografici sui dati stessi, ma a questo punto bisogna chiedersi: Cosa si intende per “*metodi crittografici*”, o meglio cos'è la *Crittografia*? La risposta a questo quesito la troveremo nel paragrafo a seguire.

3 http://banchedati.camera.it/Decretilgs/GetHtml.asp?Item=0

CAPITOLO 1
Il modello matematico

Introduzione

In questo capitolo vengono proposti alcuni dei principali modelli matematici con le loro rispettive implementazioni in codice Java. Tali strutture di carattere algebrico, serviranno nell'implementazione degli algoritmi crittografici proposti nel *capitolo 2*.

L'Algoritmo di Euclide

Come noto il metodo di Euclide consente di determinare il Massimo Comune Divisore tra due interi, ma prima di entrare nel dettaglio di tale metodologia riprendiamo il concetto stesso di divisione tra due numeri interi : siano a e b interi, con $b \neq 0$, allora esistono e sono univocamente individuati due interi q e r tali che $a = bq + r$, con $0 \leq r \leq |b|$.

Chiameremo *a dividendo*, *b divisore*, *q quoziente* ed *r resto* della divisione. A tal proposito, diremo che un numero a si dice *divisibile* per un numero $b \neq 0$, o si dice che b è un divisore di a oppure che b divide a e si scrive $b|a$, se il resto della divisione di a per b è zero. Il altre parole, a è divisibile per b se esiste un intero m tale che $a=mb$, cioè a è un multiplo intero di b.

Se $a>1$ ha soltanto divisori banali si dice un *numero irriducibile* oppure *primo*.

Come precedentemente anticipato, l'algoritmo di Euclide, consente il calcolo del MCD di due interi mediante il seguente metodo delle divisioni successive:

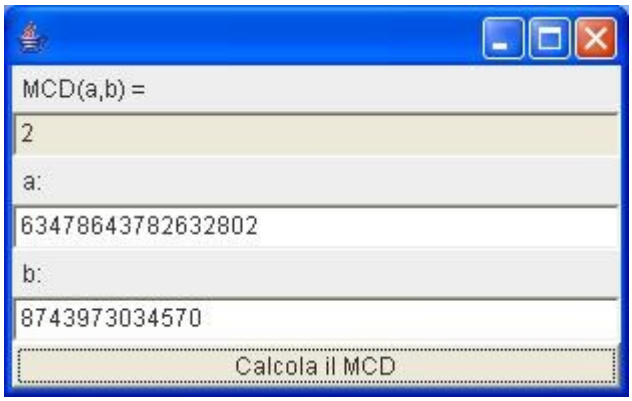
1.	$a = b\,q_1 + r_1$	$0 < r_1 < b$
2.	$b = r_1\,q_2 + r_2$	$0 < r_2 < r_1$
...	...	...
i+2	$r_i = r_{i+1}\,q_{i+2} + r_{i+2}$	$0 < r_{i+2} < r_{1+1}$
...	...	...
n-1	$r_{n-3} = r_{n-2}\,q_{n-1} + r_{n-1}$	$0 < r_{n-1} < r_{n-2}$
n	$r_{n-2} = r_{n-1}\,q_n + \boldsymbol{r_n}$	$0 < r_n < r_{n-1}$
n+1	$r_{n-1} = r_n\,q_{n+1} + 0$	

Il cui rispettivo metodo ricorsivo della classe *BigIntegerMath.java* risulta essere il seguente:

```
/**
 * Determina ricorsivamente il MCD tramite l'algoritmo Euclideo
 * @param first di tipo BigInteger, rappresenta il primo parametro
 * @param second di tipo BigInteger, rappresenta il secondo parametro
 * @return un dato di tipo BigInteger, rappresentante il MCD(first,second)
```

```
 */
public static BigInteger gcd(BigInteger first, BigInteger second) {
      //ne determina il valore assoluto nel caso i parametri siano negativi
      first=first.abs();
      second=second.abs();
      //Risolve il problema ricorsivamente
      return recurseGCD(first,second);
}//gcd

/**
 * Una versione ricorsiva dell'algoritmo Euclideo
 * @param x di tipo BigInteger
 * @parama y di tipo BigInteger
 * */
private static BigInteger recurseGCD(BigInteger x, BigInteger y) {
      if (y.equals(ZERO)) return x;
      else return recurseGCD(y,x.mod(y));
}//recurseGCD
```

la cui esecuzione nella classe *TestGCD.java* risulta essere la seguente:



il cui codice risulta essere il seguente:

```
package algebra;


import java.math.*;
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

import algebra.BigIntegerMath;;

public class TestGCD extends JFrame implements ActionListener {

   Label gcdLabel=new Label(" MCD(a,b) = ");
   TextField msg=new TextField(40);
   Label num1Label=new Label(" a: ");
   TextField entryField1=new TextField(40);
   Label num2Label=new Label(" b: ");
   TextField entryField2=new TextField(40);
   Button genGCDButton=new Button("Calcola il MCD");

   public TestGCD() {
```

```
        Panel p1=new Panel();

        p1.setLayout(new GridLayout(7,1));
        p1.add(gcdLabel);
        p1.add(msg);
        msg.setEditable(false);
        p1.add(num1Label);
        p1.add(entryField1);
        p1.add(num2Label);
        p1.add(entryField2);
        p1.add(genGCDButton);
        genGCDButton.addActionListener(this);
      add(p1);
      pack();
      show();
  }

    public void actionPerformed(ActionEvent e) {
       if (e.getSource()==genGCDButton) {
          BigInteger g=null;
          try {
             BigInteger a=new BigInteger(entryField1.getText());
             BigInteger b=new BigInteger(entryField2.getText());
      g = BigIntegerMath.gcd(a,b);
             msg.setText(g.toString());
          } catch (NumberFormatException nfe) {
             msg.setText("Non è un intero!");
      }
    }
  }

    public static void main(String[] arg){
            new TestGCD();
       }//main

}
```

L'identità di Bézout

L'algoritmo di Euclide ci offre anche un modo per provare che esiste una relazione del tipo

$$MCD(a,b) = \alpha\, a + \beta\, b$$

con α e β opportuni interi. Tale relazione si chiava *identità di Bézout*. Per dimostrare tale relazione, basta far vedere che tutti i resti delle divisioni successive si possono scrivere come combinazioni lineari di *a* e *b*. Infatti osserviamo che:

- $r_1 = a - bq_1$
- $r_2 = b - r_1 q_2$
- ...
- $r_n = r_{n-2} - r_{n-1}\, q_n$

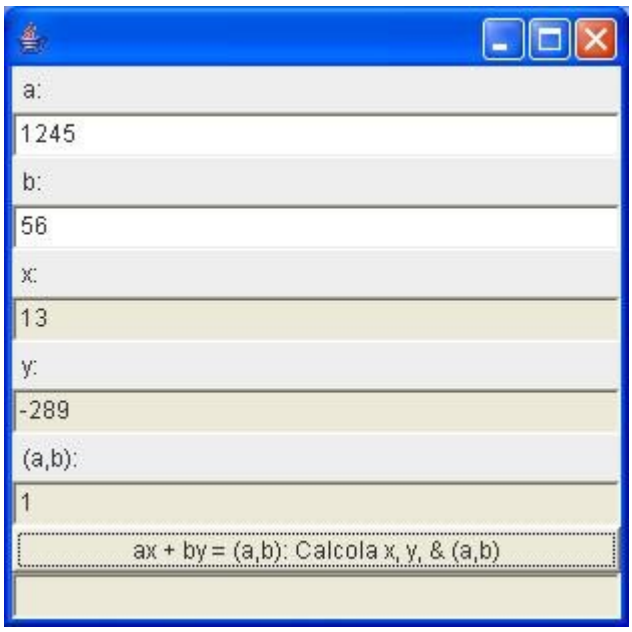
da tale asserto segue che due interi positivi *a* e *b* sono *primi* tra loro se e solo se esistono due interi α e β tali che $\alpha\, a + \beta\, b = 1$.

Il metodo della classe *BigIntegerMath.java* che calcola il *MCD(a,b)* con il metodo di Bézout risulta essere il seguente:

```
/**
```

```
 * Una versione non ricorsiva del Teorema di Euclide.
 * MCD(x, y) = (s)x + (t)y
 * @param a di tipo BigInteger positivo
 * @param b di tipo BigInteger positivo
 * @exception se i parametri <b>a</b> e <b>b</b> non sono positivi
 * @return un array di 3 BigIntegers, di cui:
 * <DIR>
 * <LI> answer[0] è il MCD(x,y)
 * <LI> answer[1] è il coefficiente s
 * <LI> answer[2] è il coefficiente t
 * </DIR>
 * */
public static BigInteger[] euclid(BigInteger a,BigInteger b) throws
IllegalArgumentException {

            //Solleva un eccezione se i parametri non sono positivi
      if (a.compareTo(ZERO)<=0||b.compareTo(ZERO)<=0) throw new
IllegalArgumentException("Euclid requires both arguments to be positive!");
            BigInteger[] answer=new BigInteger[3];

            //Imposta tutte le voci della tabella iniziale
            BigInteger r0=new BigInteger(a.toByteArray());
            BigInteger r1=new BigInteger(b.toByteArray());
            BigInteger s0=new BigInteger("1");
            BigInteger s1=new BigInteger("0");
            BigInteger t0=new BigInteger("0");
            BigInteger t1=new BigInteger("1");
            BigInteger q1=r0.divide(r1);
            BigInteger r2=r0.mod(r1);
            BigInteger s2,t2;

            //Quando r2 assume valore zero, il valore precedente è il risultato
atteso
            while (r2.compareTo(ZERO)>0) {
                  s2=s0.subtract(q1.multiply(s1));
                  s0=s1;
                  s1=s2;
                  t2=t0.subtract(q1.multiply(t1));
                  t0=t1;
                  t1=t2;
                  r0=r1;
                  r1=r2;
                  q1=r0.divide(r1);
                  r2=r0.mod(r1);
            }//while

            answer[0]=r1;
            answer[1]=s1;
            answer[2]=t1;

            return answer;

}//euclid
```

mentre la sua implementazione grafica nella classe TestEuclid.java risulta essere la seguente:




```
package algebra;

import java.math.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

import algebra.BigIntegerMath;

public class TestEuclid extends JFrame implements ActionListener {

   Label aLabel=new Label(" a: ");
   TextField entryField1=new TextField(40);
   Label bLabel=new Label(" b: ");
   TextField entryField2=new TextField(40);
   Label xLabel=new Label(" x: ");
   TextField outField1=new TextField(40);
   Label yLabel=new Label(" y: ");
   TextField outField2=new TextField(40);
   Label gLabel=new Label(" (a,b): ");
   TextField outField3=new TextField(40);
   Button computeButton=new Button("ax + by = (a,b): Calcola x, y, & (a,b)");
   TextField msg=new TextField(40);

   public TestEuclid() {

         Panel p1=new Panel();

         p1.setLayout(new GridLayout(12,1));
         p1.add(aLabel);
         p1.add(entryField1);
         p1.add(bLabel);
         p1.add(entryField2);
         p1.add(xLabel);
         p1.add(outField1);
         outField1.setEditable(false);
         p1.add(yLabel);
         p1.add(outField2);
```

```
            outField2.setEditable(false);
            p1.add(gLabel);
            p1.add(outField3);
            outField3.setEditable(false);
            p1.add(computeButton);
            computeButton.addActionListener(this);
            p1.add(msg);
            msg.setEditable(false);
          add(p1);
          pack();
          show();
  }

    public void actionPerformed(ActionEvent e) {
       if (e.getSource()==computeButton) {
          BigInteger[] answers=null;
          try {
             BigInteger a=new BigInteger(entryField1.getText());
             BigInteger b=new BigInteger(entryField2.getText());
       answers=BigIntegerMath.euclid(a,b);
             outField1.setText(answers[1].toString());
             outField2.setText(answers[2].toString());
             outField3.setText(answers[0].toString());
             msg.setText("");
          } catch (NumberFormatException nfe) {
             msg.setText(nfe.toString());
          } catch (IllegalArgumentException iae) {
             msg.setText(iae.toString());
      }
    }
  }

    public static void main(String[] arg){
             new TestEuclid();
       }//main

}
```

Equazioni lineari diofantee

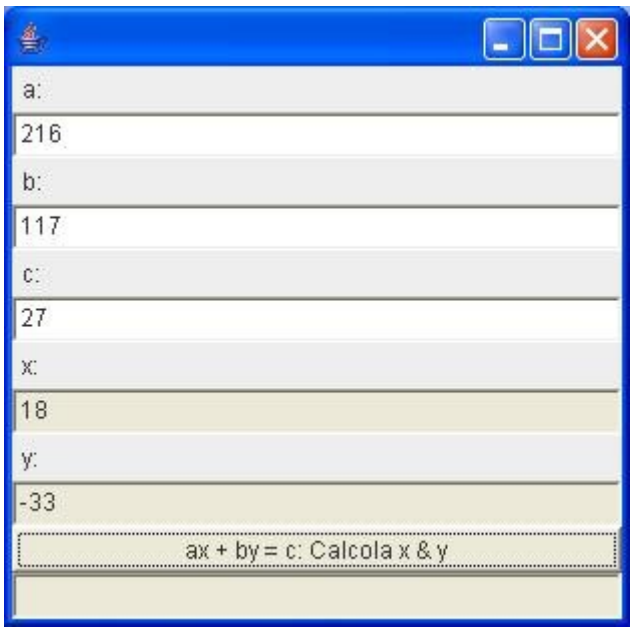
Una prima applicazione di quanto abbiamo visto in questo paragrafo è lo studio delle cosiddette *equazioni lineari diofantee*. Si tratta di equazioni del tipo $ax + by = c$, con $a, b, c \in \mathbb{Z}$ e a,b non nulli, possiede una soluzione intera (x,y) se solo se il $MCD(a,b)$ divide c.

Il metodo della classe *BigIntegerMath.java* che risolve l'equazione lineare diofantea $ax + by = c$ risulta essere il seguente:

```
/**
 * Risolve un'equazione lineare Diofantea
 * Consideriamo <b>a</b> e <b>b</b> sono interi non negativi cpn MCD=(a,b)=d.
 * Se d|c, le soluzioni intere <b>x</b> e <b>y</b> dell'equazione <i>ax + by =
c</i>
 * sono x = x0 + bn/d, y = y0 - an/d, dove x = x0, y = y0 sono la prima
soluzione.
 * Se non d|c, l'equazione non ammette soluzioni intere.
 * @param a di tipo BigInteger positivo
 * @param b di tipo BigInteger positivo
 * @param c di tipo BigInteger non negativo.
 * @exception Se <b>a</b> o <b>b</b> non sono positivi,
```

```
 * oppure se <b>c</b> è negativo, verrà restituito un
<i>IllegalArgumentException</i>.
 * Inoltre, se non d|c, verrà restituito un <i>IllegalArgumentException</i>.
 * @return determina una particolare soluzione <i> x = x' </i>,
 * <i>y = y'</i> dell'equazione <i>ax + by = c</i>, la quale verrà restituita in
un
 * array di BigIntegers.
 * l'elemento di indice 1 sarà x', e y' sarà l'elemento di indice 2.
 * Per convenzione, il MCD di <i>a</i> e <i>b</i>verrà restituito nell'elemento
di indice 0
 * */
public static BigInteger[] solveLinearDiophantine (BigInteger a, BigInteger b,
BigInteger c) throws IllegalArgumentException {

      if (a.compareTo(ZERO)<=0||b.compareTo(ZERO)<=0||c.compareTo(ZERO)<0)
            throw new IllegalArgumentException ("Tutti i coefficienti devono
essere positivi in un'equazione lineare diofantea.");
      BigInteger[] euclidAnswers=euclid(a,b);
      if (c.mod(euclidAnswers[0]).compareTo(ZERO)!=0)
            throw new IllegalArgumentException ("Non esistono soluzioni poiche'
"+euclidAnswers[0]+" non divide "+c+".");
      BigInteger[] answer=new BigInteger[3];
      BigInteger q=c.divide(euclidAnswers[0]);
      answer[0]=euclidAnswers[0];
      answer[1]=q.multiply(euclidAnswers[1]);
      answer[2]=q.multiply(euclidAnswers[2]);

      return answer;

}//solveLinearDiophantine
```

mentre la sua implementazione grafica nella classe TestDiophantine.java risulta essere la seguente:




```
package algebra;

import java.math.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
import algebra.BigIntegerMath;

public class TestDiophantine extends JFrame implements ActionListener {

   Label aLabel=new Label(" a: ");
   TextField entryField1=new TextField(40);
   Label bLabel=new Label(" b: ");
   TextField entryField2=new TextField(40);
   Label cLabel=new Label(" c: ");
   TextField entryField3=new TextField(40);
   Label xLabel=new Label(" x: ");
   TextField outField1=new TextField(40);
   Label yLabel=new Label(" y: ");
   TextField outField2=new TextField(40);
   Button computeButton=new Button("ax + by = c: Calcola x & y");
   TextField msg=new TextField(40);

   public TestDiophantine() {

         Panel p1=new Panel();

         p1.setLayout(new GridLayout(12,1));
         p1.add(aLabel);
         p1.add(entryField1);
         p1.add(bLabel);
         p1.add(entryField2);
         p1.add(cLabel);
         p1.add(entryField3);
         p1.add(xLabel);
         p1.add(outField1);
         outField1.setEditable(false);
         p1.add(yLabel);
         p1.add(outField2);
         outField2.setEditable(false);
         p1.add(computeButton);
         computeButton.addActionListener(this);
         p1.add(msg);
         msg.setEditable(false);
       add(p1);
       pack();
       show();
  }

   public void actionPerformed(ActionEvent e) {
      if (e.getSource()==computeButton) {
         BigInteger[] answers=null;
         try {
            BigInteger a=new BigInteger(entryField1.getText());
            BigInteger b=new BigInteger(entryField2.getText());
            BigInteger c=new BigInteger(entryField3.getText());
    answers=BigIntegerMath.solveLinearDiophantine(a,b,c);
            outField1.setText(answers[1].toString());
            outField2.setText(answers[2].toString());
            msg.setText("");
         } catch (NumberFormatException nfe) {
            msg.setText(nfe.toString());
         } catch (IllegalArgumentException iae) {
            msg.setText(iae.toString());
```

```
    }
  }
}

  public static void main(String[] arg){
          new TestDiophantine();
     }//main

}
```

Congruenze

Sia n un intero positivo fissato. Si dice *relazione di congruenza* modulo n la relazione in $\mathbb{Z}$ definita nel modo seguente: $a \equiv b \ (mod\ n)$ se e solo se esiste un intero h tale che $a-b = nh$, ossia:

$$a \equiv b \ (mod\ n) \text{ se e solo se } n \mid (a-b)$$

e si dice in tal caso che a è *congruo* a b modulo n.

Risulta facile verificare che la congruenza modulo un qualunque intero positivo n è una *relazione di equivalenza* e per tale motivo vengono definite come *classi dei resti modulo n*, poiché ogni intero a è congruo modulo n ad un unico intero r (il resto) tale che $0 \leq r < n$.

Nella dimostrazione delle strutture algebriche a seguire, denoteremo con $\mathbb{Z}_n$ l'*insieme quoziente* di $\mathbb{Z}$ rispetto alla congruenza modulo n, cioè $\mathbb{Z}_n$ consiste delle n classi distinte dei resti modulo n :

$$\mathbb{Z}_n = \{\bar{0}, \bar{1}, \bar{2}, \ldots, (\bar{n-2}), (\bar{n-1})\} = \mathbb{Z} \ / \equiv_n$$

Tale passaggio dall'infinito al finito risulta utile quando si vogliono effettuare calcoli o verifiche con il calcolatore, il quale, come è noto lavora soltanto in termini finiti!

Di seguito viene proposto il metodo della classe *BigIntegerMath.java* che determina il modulo (il resto più piccolo) tra due interi:

```
/**
 * Consideriamo <b>b</b> come un intero ed <b>m</b> come un intero positivo.
 * Tutti gli interi congruenti <b>b modulo m</b> sono chiamati
 * <b>resti residui b modulo m</b>.
 * Il più piccolo resto non negativo, <i>lnr</i> (dall'inglese least nonnegative
residue),
 * di <i>b modulo m</i> è il più piccolo intero non negativo congruo <i>b modulo
m</i>.
 * Esempio : il lnr di 29 modulo 13 è 3. Questi è, 3 congruo 29 (mod 13), and 3
 * è il più piccolo intero non negativo congruo 29 modulo 13.
 *
 * La classe BigInteger class fornisce il metodo mod(),
 * il quale può restituire un valore negativo se la divisione è negativa.
 * Correggeremo questo valore aggiungendo a questi il valore del modulo.
 * Pertano, in questa circostanza non sono consentiti moduli negativi.
 * @param b di tipo BigInteger
 * @param m di tipo BigInteger positivo
 * @exception m deve essere positivo
 * @return un valore BigInteger rappresentate il più piccolo resto non negativo,
<i>lnr</i> (dall'inglese least nonnegative residue),
 * di <i>b modulo m</i>
 * */
public static BigInteger lnr(BigInteger b, BigInteger m) {
      if (m.compareTo(ZERO)<=0)
            throw new IllegalArgumentException("Il modulo deve essere
```

```
positivo.");
```
BigInteger answer=b.mod(m);

```
      return (answer.compareTo(ZERO)<0)?answer.add(m):answer;
}//lnr
```

Congruenze lineari

Si definisce *congruenza lineare* oppure *equazione congruenziale* nell'incognita x un'equazione della forma $ax \equiv b \pmod n$ con a,b in $\mathbb{Z}$ e n intero positivo maggiore di 1. Se tale equazione ammette soluzione, diremo che questi è *risolubile* o *compatibile* in particolare:

1. essa ammette soluzione se e solo se $d = MCD(a,n)$ divide b;
2. se la congruenza è compatibile, indicata con x_0 una sua soluzione, tutte e sole le soluzioni sono della forma : $x_0 + hm$, con $h \in \mathbb{Z}$, dove $m=n|d$. Tra queste le soluzioni x_0 , x_0+m, x_0+2m, ... , $x_0+(d-1)m$ sono tutte non congruenti tra di loro e ogni altra è congruente ad una di queste. Quindi la congruenza ammette esattamente d soluzioni non congruenti modulo n

da quest'ultimo punto segue che se a è *relativamente primo* con n, ovvero $MCD(a,n)=1$, allora la congruenza $ax \equiv b \pmod n$ ammette un'unica soluzione modulo n.

In definitiva, stando alle metodologie fin'ora esposte, per risolvere una congruenza lineare del tipo $ax \equiv b \pmod n$ occorre, risolvere un'equazione diofantea lineare del tipo $ax + ny = b$, eseguendo i seguenti punti:

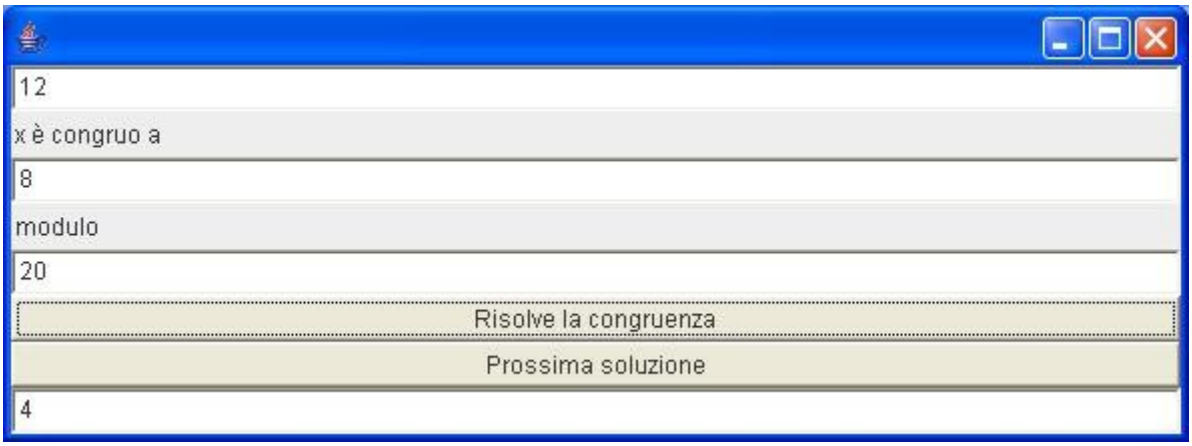
1. calcolare $d = MCD(a,n)$, il che ha complessità $O(\log^3 a)$;
2. dividere b per d, il che ha complessità $O(\log a \log b)$;
3. a questo punto possiamo decidere se l'equazione sia o meno risolubile, verificando se b sia o meno divisibile per d. Dunque la complessità dell'algoritmo che *decide* se l'equazione è risolubile ha complessità $O(\log^3 a + \log a \log b)$;
4. quanto a risolvere effettivamente l'equazione diofantea lineare $ax + ny = b$ e dunque una soluzione x_o di $ax \equiv b \pmod n$, occorre calcolare un'identità di Bézout per $MCD(a,n)$ e moltiplicarla per b/d.

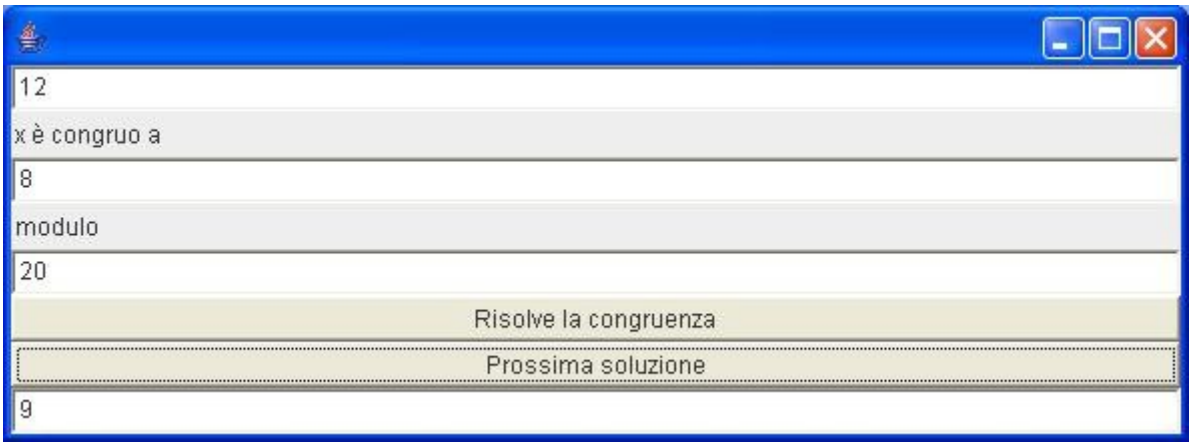
Questi quattro punti sono stati tradotti nel seguente metodo della classe *BigIntegerMath.java* :

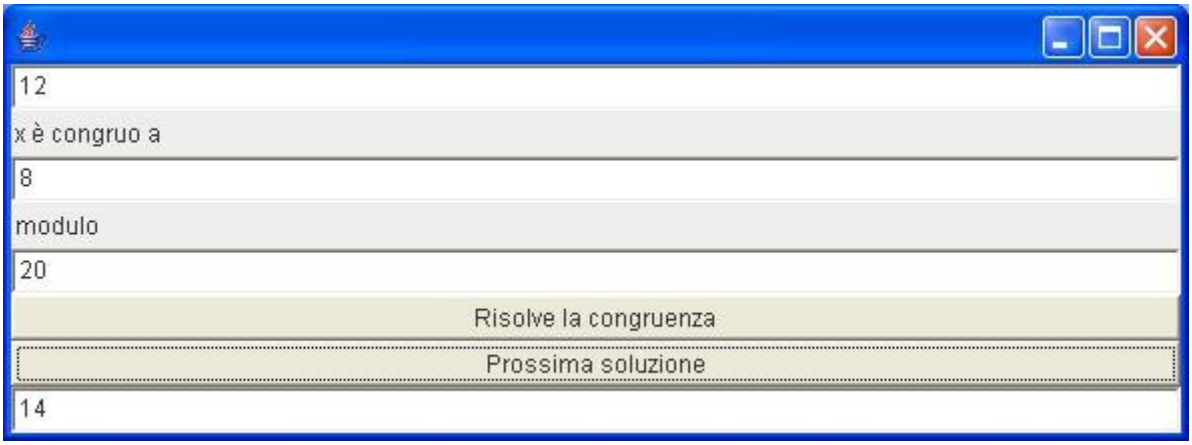
```
/**
 * Risove una congruenza lineare del tipo a<i>x</i> congruo b (mod m).
 * La soluzione ti tale equazione avviene mediante la risuluzione dell'equazione
 * lineare diofantea associata, ovvero ax + my = b t.c. a = lnr(a,m) e b =
lnr(b,m)
 * @param a di tipo BigInteger, rappresenta il coefficiente della variabile
<i>x</i>
 * @param b di tipo BigInteger, rappresenta il termine noto dell'equazione
 * @param m di tipo BigInteger, rappresenta il modulo
 * @return un'array di BigInteger nel quale:
 * <DIV>
 * <LI> nella posizione 0 c'è il MCD(a,b).
 * <LI> nelle restanti posizioni ci sono le eventuali soluzioni.
 * </DIV>
 * */
public static BigInteger[] solveLinearCongruence(BigInteger a, BigInteger b,
BigInteger m) {

      BigInteger[] answers=solveLinearDiophantine(lnr(a,m),m,lnr(b,m));
```

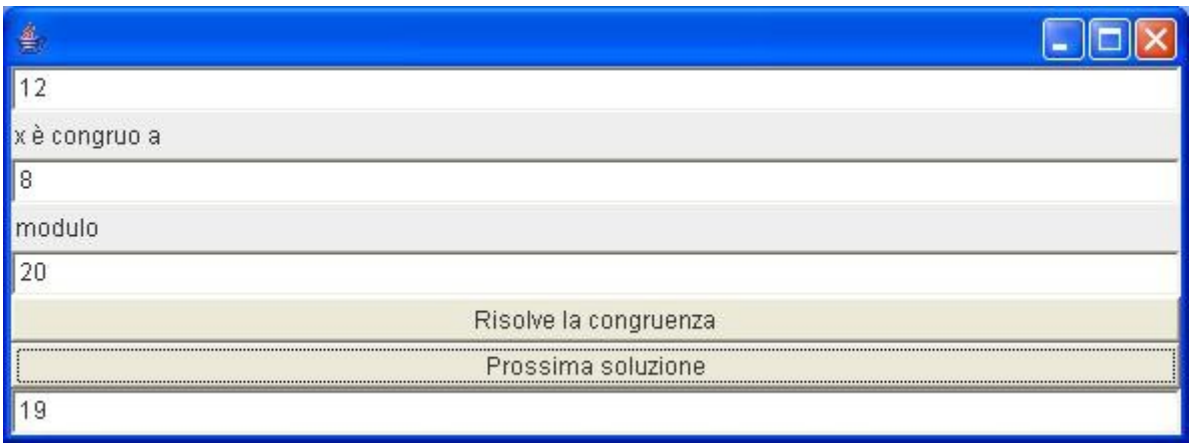
```
        return answers;
}//solveLinearCongruence
```

la cui implementazione grafica è stata definita nella classe *TestLinearCongruence.java* :










```
package algebra;

import java.math.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

import algebra.BigIntegerMath;

public class TestLinearCongruence extends JFrame implements ActionListener {

        private static BigInteger t;
        BigInteger[] ans;
        BigInteger a;
        BigInteger b;
        BigInteger m;
        TextField aBox=new TextField(20);
        Label Label1=new Label("x è congruo a");
        TextField bBox=new TextField(20);
        Label Label2=new Label("modulo");
        TextField mBox=new TextField(20);
        Button solveButton=new Button("Risolve la congruenza");
        Button nextButton=new Button("Prossima soluzione");
        TextField msgBox=new TextField(80);

        public TestLinearCongruence() {

                Panel p1=new Panel();

                p1.setLayout(new GridLayout(8,1));
                p1.add(aBox);
                p1.add(Label1);
                p1.add(bBox);
                p1.add(Label2);
                p1.add(mBox);
                p1.add(solveButton);
                solveButton.addActionListener(this);
                p1.add(nextButton);
                nextButton.addActionListener(this);
                nextButton.setEnabled(false);
                p1.add(msgBox);
                add(p1);
                pack();
                show();
        }
        public void actionPerformed(ActionEvent e) {
```

```
            if (e.getSource()==solveButton) {
              try {
                t=new BigInteger("0");
                a=new BigInteger(aBox.getText());
                b=new BigInteger(bBox.getText());
                m=new BigInteger(mBox.getText());
                ans=BigIntegerMath.solveLinearCongruence(a,b,m);
                msgBox.setText(ans[1].mod(m).toString());
                nextButton.setEnabled(true);
              } catch (NumberFormatException nfe) {
                nextButton.setEnabled(false);
                msgBox.setText(nfe.toString());
              } catch (IllegalArgumentException iae) {
                nextButton.setEnabled(false);
                msgBox.setText(iae.toString());
              }
            } else if (e.getSource()==nextButton) {
              t=t.add(BigIntegerMath.ONE).mod(m);

msgBox.setText((ans[1].add(t.multiply(m).divide(ans[0]))).mod(m).toString());
            }
          }

          public static void main(String[] arg){
                    new TestLinearCongruence();
            }//main
}
```

Sistemi di Congruenze lineari

Ovviamente possiamo considerare lo studio di equazioni di congruenze di sistemi lineari, aventi determinante dei coefficienti delle incognite non nullo. In prima istanza considereremo sistemi di equazioni tutti ridotti allo stesso modulo, per poi considerare sistemi di equazioni con moduli diversi (*Teorema Cinese di Resti*).

Consideriamo il seguente sistema lineare di congruenze:

$x+4y \equiv 3 \ (mod\ 5)$

$2x+2y \equiv 2 \ (mod\ 5)$

questi può essere scritto nella forma $AX \equiv B \ (mod\ 5)$ tale che:

$$A=\begin{pmatrix}1 & 4\\ 2 & 2\end{pmatrix} \qquad X=\begin{pmatrix}x\\ y\end{pmatrix} \qquad B=\begin{pmatrix}3\\ 2\end{pmatrix}$$

Per tale tipo di struttura, è stato creato un package apposito, denominato *matrici* all'interno del quale vengono racchiuse diverse classi dedite appunto alla risoluzione di sistemi di congruenze lineari; tra queste citiamo :

- *ModMatrix.java* : classe che consente di gestire matrici di *r* righe e *c* colonne modulo *m*, quindi anche matrici rettangolari su cui eseguire operazioni come: somma, differenza, moltiplicazione sia tra matrici che per uno scalare. Alcuni dei metodi proposti potranno sollevare delle eccezioni in merito alle dimensioni non conformi delle matrici modulari proposte, tutte gestite dalla classe *MatricesNonConformableException.java.* .
  Il codice sorgente viene fornito in seguito:

```
package matrici;

import java.math.*;
```

```
import java.security.*;

import algebra.BigIntegerMath;

/**
 * @author Giuseppe Specchio
 * @date 25/05/2008
 * Elabora Matrici di <i>r</i> righe e <i>c</i> colonne in algebra modulare.
 * */

public class ModMatrix {

  //Un oggetto ModMatrix è un array 2D (matrice) di BigIntegers
  BigInteger[][] array;

  //Numero di righe e colonne
  int numRows, numCols;

  //Il modulo della matrice ModMatrix
  BigInteger modulus;

  /**
   * Costruttore di default
   */
  public ModMatrix() {}

  /**
   * Crea una matrice con valori casuali caratterizzata da <i>r</i> righr e
<i>c</i> colonne,
   * oppure, viene instanziata una matrice con tutti zero.
   * La matrice ha indice iniziale 1,1, quindi non vengono utilizzate righe o
colonne di indice zero.
   * @param r di tipo int, rappresenta il numero di righe della matrice
   * @param c di tipo int, rappresenta il numero di colonne della matrice
   * @param makeZero di tipo boolean, assume valore true se si vuole una matrice
con tutti zero, diversamente genererà una matrice con valori random
  */
  public ModMatrix(int r,int c,BigInteger m,boolean makeZero) {
          SecureRandom sr=new SecureRandom();
          modulus=m;
          array=new BigInteger[r+1][c+1];
          numRows=r;
          numCols=c;
          for (int i=0;i<r;i++) {
              for (int j=0;j<c;j++) {
                  //Se makeZero è settato a true, crea una matrice con tutti zero
                  if (makeZero) array[i+1][j+1]=new BigInteger("0");
                  //otherwise, make matrix with random entries
                  else array[i+1][j+1]=new
BigInteger(modulus.bitLength(),sr).mod(modulus);
        }
      }
  }

  /**
   * Crea una matrice da un array <i>a</i>.
   * Se l'array <i>a</i> non contiene sufficienti valori
   * per riempire la matrice, le rimanenti celle vengono
   * riempite con degli zero.
   * @param r di tipo int, rappresenta il numero di righe della matrice
   * @param c di tipo int, rappresenta il numero di colonne della matrice
```

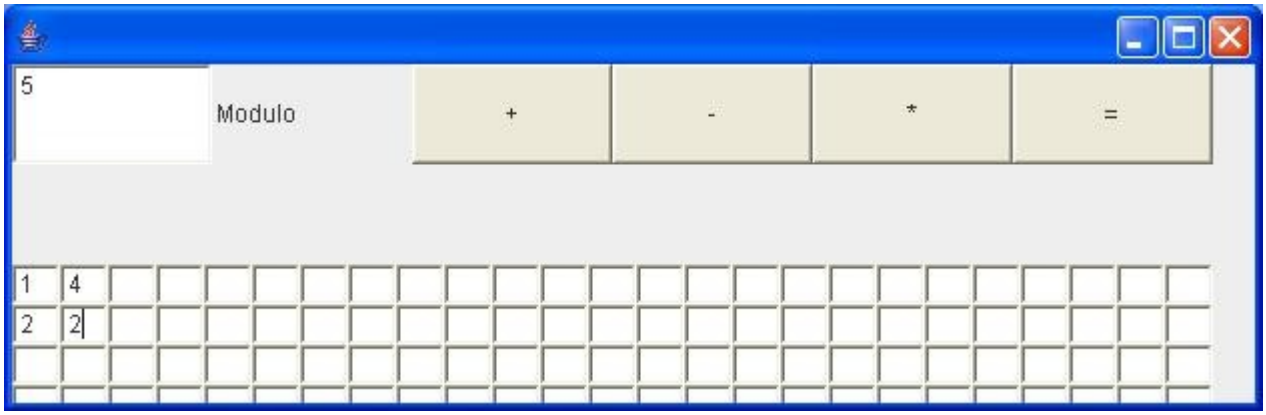
```
   * @param a di tipo BigInteger[], rappresenta l'array da trasformare in
matrice
   * @param m di tipo BigInteger, rappresenta il modulo
   * */
  public ModMatrix(int r,int c,BigInteger[] a, BigInteger m) {
         modulus=m;
         //Ottimizza le dimensioni della matrice, in modo tale da poter evitare
di gestire
         //le righe o le colonne di indice zero
         array=new BigInteger[r+1][c+1];
         numRows=r;
         numCols=c;
         for (int i=0;i<r;i++) {
             for (int j=0;j<c;j++) {
                 int pos=i*c+j;
                 //Carica i valori dall'array alla madrice in modulo <i>m</i>
                 if (pos<a.length&&a[pos]!=null)
                   array[i+1][j+1]=BigIntegerMath.lnr(a[pos],m);
                 else//le rimanenti celle vengono impostate a zero
                   array[i+1][j+1]=new BigInteger("0");
       }
     }
 }

  /**
   * Effettua la copia di un'altra Matrice in algebra modulare
   * @param m di tipo ModMatrix, rappresenta la sorgente della copia
   * */
  public ModMatrix(ModMatrix m) {
         array=new BigInteger[m.numRows+1][m.numCols+1];
         numRows=m.numRows;
         numCols=m.numCols;
         modulus=m.modulus;
         for (int i=1;i<=m.numRows;i++) {
             for (int j=1;j<=m.numCols;j++) {
                 array[i][j]=new BigInteger(m.array[i][j].toString());
       }
     }
 }

  /**
   * Restituisce il numero di righe
   * @return un valore di tipo intero
   * */
  public int rows() {
        return numRows;
 }

  /**
   * Restituisce il numero di colonne
   * @return un valore di tipo intero
   * */
  public int columns() {
        return numCols;
 }

  /**
   * Restituisce il modulo della matrice
   * @return un valore di tipo BigInteger
```

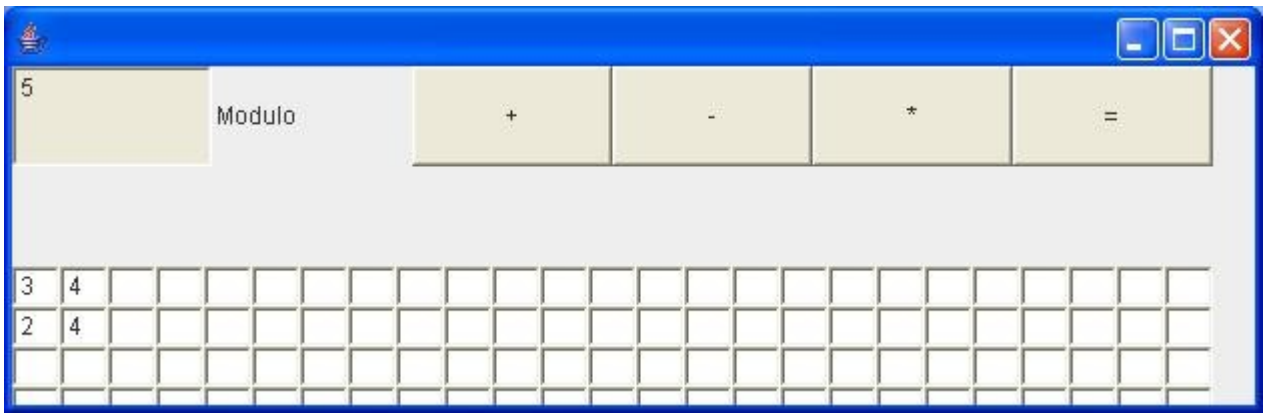
```
   * */
  public BigInteger getModulus() {
        return modulus;
 }

  //Allows one to retrieve an element.
  /**
   * restituisce l'elemento di coordinate <i>(r,c)</i>
   * @param r di tipo int, rappresenta il numero di righe della matrice
   * @param c di tipo int, rappresenta il numero di colonne della matrice
   * @return un valore di tipo BigInteger
   * */
  public BigInteger getElement(int row,int column) {
        return array[row][column];
 }

  /**
   * Determina il congruo di un determinato valore, da memorizzare in una data
posizione della matrice
   * @param r di tipo int, rappresenta il numero di righe della matrice
   * @param c di tipo int, rappresenta il numero di colonne della matrice
   * @param value di tipo BigInteger, rappresenta il valore di cui bisogna
calcolarne la congruenza e
   * memorizzarlo in posizione <i>(r,c)</i>
   * */
  public void setElement(int row,int column,BigInteger value) {
        array[row][column]=BigIntegerMath.lnr(value,modulus);
 }

  /**
   * Effettua la somma alla matrice corrente la matrice modulare <i>m</i>
   * @param m di tipo ModMatrix, rappresenta il secondo addendo, ovvero la
matrice da aggiungere alla matrice corrente
   * @exception Le matrici devono avere la stessa dimensione ed essere gestiste
con lo stesso modulo
   * @return una matrice di tipo ModMatrix
   * */
  public ModMatrix add(ModMatrix m) throws MatricesNonConformableException {
     ModMatrix result;
        //Le matrici devono aveve la stessa dimensione ed essere gestiste con
lo stesso modulo
        if (!modulus.equals(m.modulus))
            throw new MatricesNonConformableException("Questa matrici non
possono essere sommate, poiché hanno moduli differenti.");
        if (numRows==m.numRows&&numCols==m.numCols) {
                //Istanzia una nuova ModMatrix per effettura la somma con una
matrice nulla
                result=new ModMatrix(numRows,numCols,modulus,true);
                //Effettua la somma degli elementi di coordinate (i,j) e ne
determina il modulo
                for (int i=1;i<=numRows;i++) {
                    for (int j=1;j<=numCols;j++) {
                        result.array[i][j]=BigIntegerMath.lnr(array[i]
[j].add(m.array[i][j]),modulus);
          }
        }
        } else
            throw new MatricesNonConformableException("Questa matrici non
possono essere sommate, poiché hanno dimensioni differenti.");
        return result;
```

```
 }

  /**
   * Effettua la sottrazione alla matrice corrente della matrice modulare <i>m</
i>
   * @param m di tipo ModMatrix, rappresenta il secondo parametro, ovvero la
matrice da sottrarre alla matrice corrente
   * @exception Le matrici devono avere la stessa dimensione ed essere gestiste
con lo stesso modulo
   * @return una matrice di tipo ModMatrix
   * */
  public ModMatrix subtract(ModMatrix m) throws MatricesNonConformableException
{
     //moltiplica la seconda matrice per lo sclare -1 e ne determina la somma
con la matrice corrente
     return this.add(m.multiply(new BigInteger("-1")));
 }

  /**
   * Effettua la moltiplicazione alla matrice corrente di <i>r</i> righe e
<i>c</i> colonne
   * con la matrice modulare <i>m</i> di <i>m</i> righe e <i>r</i> colonne
   * @param m di tipo ModMatrix, rappresenta il secondo parametro, ovvero la
matrice da moltiplicare alla matrice corrente
   * @exception La seconda matrice deve avere una quantità di righe pari alla
quantità di colonne della matrice corrente,
   * nonchè devono utilizzare lo stesso modulo
   * @return una matrice di tipo ModMatrix
   * */
  public ModMatrix multiply(ModMatrix m) throws MatricesNonConformableException
{
         ModMatrix result;
         //Entrambi le matrici devo usare lo stesso modulo
         if (!modulus.equals(m.modulus))
             throw new MatricesNonConformableException("Queste matrici non
possono essere moltiplicate, in quanto hanno un modulo diverso.");
         //se il numero di righe della seconda matrice è uguale al numero
         //di colonne della prima matrice, è possibile effettuare la
moltiplicazione
         if (m.numRows==numCols) {
                  result=new ModMatrix(numRows,m.numCols,modulus,true);
                  //Sposta sotto le righe nel ciclo esterno
                  for (int i=1;i<=numRows;i++) {
                      //Moltiplica la i-esima riga della prima matrice
                    //con la j-esima colonna della seconda matrice
                      for (int j=1;j<=m.numCols;j++) {
                          //Setta gli elementi di posizione i e colonna j a zero
                          result.array[i][j]=new BigInteger("0");
                          //l'elemento i, j-esima è entrata somma di i, k-esimo
valore di 1a volta
                          //k, j-esima valore del 2 ° per tutti i k
                          for (int k=1;k<=m.numRows;k++)
                             result.array[i]
[j]=BigIntegerMath.lnr(result.array[i][j].add(array[i][k].multiply(m.array[k]
[j])),modulus);
           }
         }
         } else
             throw new MatricesNonConformableException("Questa matrici non
possono essere moltiplicate!");
```

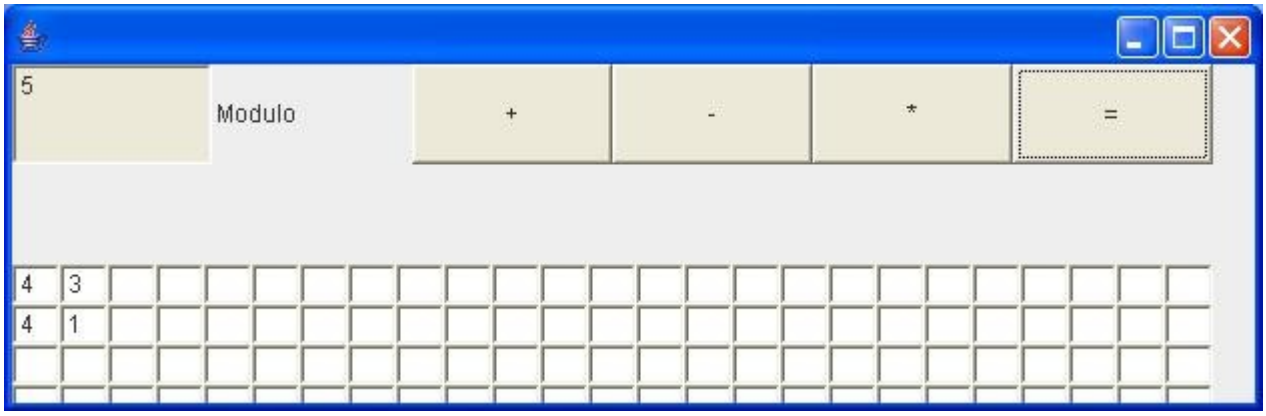
```
        return result;
 }

  //Multiplies a matrix by a scalar.
  /**
   * Moltiplica la matrice corrente per uno scalare
   * @param param di tipo BigInteger, rappresenta lo scalare
   * @return una matrice di tipo ModMatrix
   * */
  public ModMatrix multiply(BigInteger scalar) {
        ModMatrix result=new ModMatrix(numRows,numCols,modulus,true);
        for (int i=1;i<=numRows;i++)
           for (int j=1;j<=numCols;j++)
                //Moltiplica gli elementi (i,j) per lo scalare
                result.array[i][j]=BigIntegerMath.lnr(array[i]
[j].multiply(scalar),modulus);
        return result;
 }

  /**
   * Visualizza la matrice.
   * */
  public void display() {
        for (int i=1;i<=numRows;i++) {
           for (int j=1;j<=numCols;j++) {
              System.out.print(array[i][j]+"\t");
      }
           System.out.println();
    }
 }

}
```

la cui implementazione grafica è stata definita nella classe TestModMatrixCalculator.java :

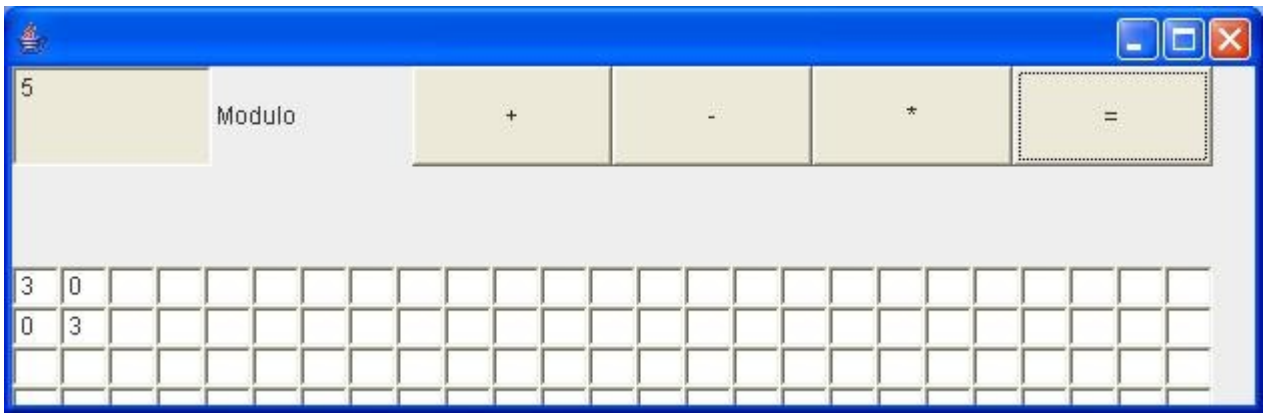




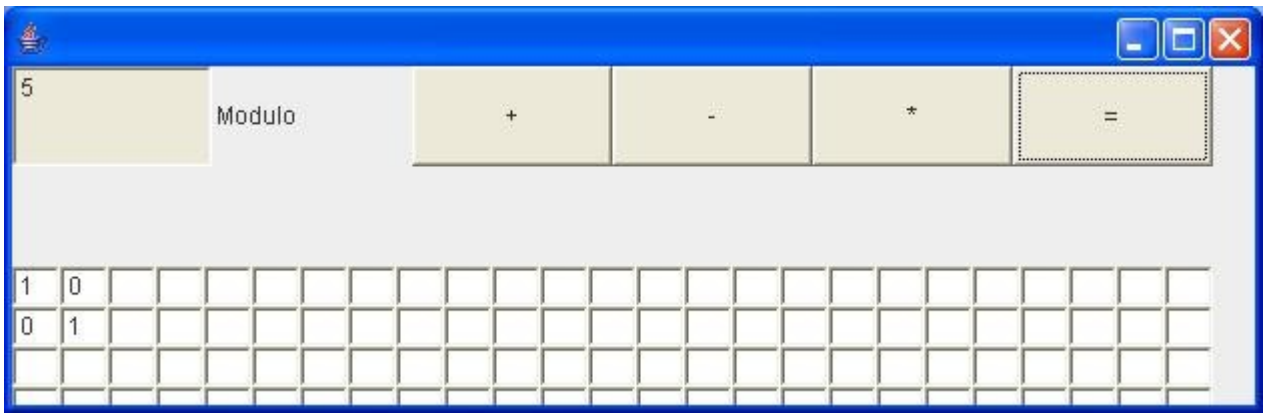
- Somma



- Differenza



- Prodotto



- *ModSquareMatrix.java* : estende la classe *ModMatrix.java* definita con metodi appropriati per la determinazione della matrice inversa e l'eliminazione di Gauss moulo *n*. Nel caso in cui venisse sollevata un'eccezione con quest'ultimo metodo, vengono attivate le classi *SingularMatrixException.java* o quella di default *ArithmeticException*in, la cui attivazione varia in base alla tipologia di errore.

```
package matrici;


import java.math.BigInteger;
import java.security.*;
import java.util.*;

import algebra.BigIntegerMath;

/**
 * Estende la classe ModMatrix.java definita con metodi
 * appropriati per la determinazione della matrice inversa e
 * l'eliminazione di Gauss moulo n.
 * */
public class ModSquareMatrix extends ModMatrix {

	/**
	 * Costruttore.
	 * Crea una matrice quadrata con valori casuali,
	 * oppure crea una matrice con tutti zero.
	 * @param s di tipo int, rappresenta la dimensione della matrice
	 * @param m di tipo BigInteger, rappresenta il modulo
	 * @param makeZero di tipo boolean, se true si ottiene una matrice di
dimensione <i>s</i> con tutti zero
	 * @param makeInvertible di tipo boolean, se true si ottiene una matrice
di dimensione <i>s</i> invertibile
	 * */
	public ModSquareMatrix(int s,BigInteger m,boolean makeZero,boolean
makeInvertible) throws MatricesNonConformableException {

		//invoca il costruttore della superclasse ModMatrix, istanziando una
matrice con tutti zero o con valori casuali
		super(s,s,m,makeZero);

		//Una mtrice nulla non è invertibile
		if (makeZero&&makeInvertible)
			throw new IllegalArgumentException ("Matrice nulla, non
inveribile!");

		//Istanzia una matrice invertibile con valori casuali
		if (makeInvertible) {
```

```
            Random r=new Random();
            SecureRandom sr=new SecureRandom();
            boolean done=false;
            //Resta nel ciclo finchè la matrice non è invertibile
            while (!done) {
               try {
                  //prova ad ottenere una matrice inversa
                    //laddove non ci riuscisse solleva un'eccezione
                  this.inverse();
                  done=true;
               } catch (SingularMatrixException sme) {
                  //Cambia i valori casuali della matrice
                  int row=Math.abs(r.nextInt())%numRows+1;
                  int col=Math.abs(r.nextInt())%numCols+1;
                  BigInteger value=new
BigInteger(modulus.bitLength(),sr).mod(modulus);
                  this.setElement(row,col,value);
               } catch (ArithmeticException ae) {
                  //Cambia i valori casuali della matrice
                  int row=Math.abs(r.nextInt())%numRows+1;
                  int col=Math.abs(r.nextInt())%numCols+1;
                  BigInteger value=new
BigInteger(modulus.bitLength(),sr).mod(modulus);
                  this.setElement(row,col,value);
         }
       }
     }
  }

    /**
     * Crea una matrice quadrata da un array
     * @param i di tipo int, rappresenta la dimensione
     * @param a di tipo BigInteger[], rappresenta l'array sorgente
     * @param m di tipo BigInteger, rappresenta il modulo
     * */
    public ModSquareMatrix(int s,BigInteger[] a,BigInteger m) {
          super(s,s,a,m);
  }

    /**
     * Crea una copia della matrice
     * @param m di tipo ModSquarematrix, rappresenta la matrice da copiare
     * */
    public ModSquareMatrix(ModSquareMatrix m) {
          array=new BigInteger[m.numRows+1][m.numCols+1];
          numRows=m.numRows;
          numCols=m.numCols;
          modulus=m.modulus;
          for (int i=1;i<=m.numRows;i++) {
              for (int j=1;j<=m.numCols;j++) {
                  array[i][j]=new BigInteger(m.array[i][j].toString());
       }
     }
  }

    /**
     *Metodo che usa l'eliminazione di Gauss per risolvere il sistema AX=B mod m
     *A è la matrice quadrata sulla quale viene invocata il metodo
     *B è la matrice dei termini noti
```

```
     *X sono le soluzioni
     *@param constants di tipo ModMatrix, rappresenta i termini noti
     * */
    public ModMatrix gaussianSolve(ModMatrix constants) throws
MatricesNonConformableException,SingularMatrixException {
          //il metodo funziona solo se il modulo è primo
       if (!modulus.isProbablePrime(16))
         throw new IllegalArgumentException("L'eliminazione di Gauss richiede che
il metodo sia primo!");
       //Copia la matrice per le modifiche
       ModSquareMatrix mat=new ModSquareMatrix(this);
    ModMatrix b;

       //Verifica se la matrice è quadrata
       if (constants instanceof ModSquareMatrix)
         b=new ModSquareMatrix((ModSquareMatrix)constants);
       else
         b=new ModMatrix(constants);

       //Verifica se le matrici sono compatibili in termini di grandezza
       if (b.numRows!=mat.numRows)
         throw new MatricesNonConformableException("Le matrici hanno differenti
quantità di matrici!");

       //Lavora sulle righe, partendo dalla prima riga
       int currentRow=1;
       while (currentRow<=mat.numRows) {
          int i=currentRow;
          //Verifica se gli elementi sulla digonale non siano nulli, laddove
possibile effettua lo scambio di righe
          while (i<=mat.numRows&&mat.array[i]
[currentRow].equals(BigIntegerMath.ZERO)) i++;
          if (i>mat.numRows)
              throw new SingularMatrixException("Rilevata dispenza lineare!");
          //Scambia con una riga che non ha zero sulla diagonale
          if (currentRow!=i) swapRows(mat,b,currentRow,i);
          //Processo di eliminazione
      i=1;
          //Moltiplica ogni riga per un proprio scalare
          while (i<=mat.numRows) {
             if (i!=currentRow) {
                BigInteger scalar=mat.array[i][currentRow];
                if (!scalar.equals(BigIntegerMath.ZERO)) {
                   multiplyRow(mat,b,i,mat.array[currentRow][currentRow]);
          multiplyRow(mat,b,currentRow,scalar);
                   //sostituisce la riga i con la riga i meno l'elemento in
diagonale
          subtractRow(mat,b,i,currentRow);
         }
       }
       i++;
     }
     currentRow++;
    }
       //Ora, produce l'elemento 1 sulla diagonale principale moltiplicando per
il suo inverso
       for (int index=1;index<=mat.numRows;index++) {
          multiplyRow(mat,b,index,mat.array[index][index].modInverse(modulus));
    }
```

```
      return b;
  }

    /**
     * Questo metodo viene attivato nel caso in cui venga invocata una matrice
quadrata
     * @param constants di tipo ModSquareMatrix, rappresenta la matrice dei
coefficienti quadrata
     * */
    public ModSquareMatrix gaussianSolve(ModSquareMatrix constants) throws
MatricesNonConformableException,SingularMatrixException {
      return (ModSquareMatrix) gaussianSolve((ModMatrix)constants);
  }

    /**
     * Usata dal metodo gaussianSolve per moltiplicare una riga per uno scalare
     * @param mat di tipo ModSquareMatrix, rappresenta la matrice quadrata
     * @param b di tipo ModMatrix, rappresenta il vettore dei termini noti
     * @param i di tipo int, rappresenta il numero di riga
     * @param scalar di tipo BigInteger, rappresenta lo scalare per cui
moltiplicare la riga <i>i</i>
     * */
    private void multiplyRow(ModSquareMatrix mat,ModMatrix b,int i,BigInteger
scalar) {

        //Moltiplica la riga i-esima per lo scalare
      for (int k=1;k<=mat.numCols;k++)
        mat.array[i][k]=BigIntegerMath.lnr(mat.array[i]
[k].multiply(scalar),mat.modulus);
      for (int k=1;k<=b.numCols;k++)
        b.array[i][k]=BigIntegerMath.lnr(b.array[i]
[k].multiply(scalar),mat.modulus);
  }

    /**
     * Usato dal metodo gaussianSolve per sottrarre righe
     * @param mat di tipo ModSquareMatrix, rappresenta la matric quadrata
     * @param b di tipo ModMatrix, rappresenta la matrice dei termini noti
     * @param i di tipo int, rappresenta l'indice della prima riga su cui
applicare la sottrazione
     * @param j di tipo int, rappresenta l'indice della prima riga su cui
applicare la sottrazione
     * */
    private void subtractRow(ModSquareMatrix mat,ModMatrix b,int i,int j) {

        //Sottrae la riga j dalla riga i, sostituendo il risultato in i
      for (int k=1;k<=mat.numCols;k++)
        mat.array[i][k]=BigIntegerMath.lnr(mat.array[i][k].subtract(mat.array[j]
[k]),mat.modulus);
      for (int k=1;k<=b.numCols;k++)
        b.array[i][k]=BigIntegerMath.lnr(b.array[i][k].subtract(b.array[j]
[k]),mat.modulus);
  }

    //Used by gaussianSolve to swap two rows
    /**
     * Usato dal metodo gaussianSolve per smabiare le righe
     * @param mat di tipo ModSquareMatrix, rappresenta la matric quadrata
     * @param b di tipo ModMatrix, rappresenta la matrice dei termini noti
     * @param r1 di tipo int, rappresenta l'indice della prima riga da scambiare
```

```
con r2
     * @param r2 di tipo int, rappresenta l'indice della prima riga da scambiare
con r1
     * */
    private void swapRows(ModSquareMatrix mat,ModMatrix b,int r1,int r2) {
    BigInteger temp;
       for (int j=1;j<=mat.numCols;j++) {
          temp=mat.array[r1][j];
          mat.array[r1][j]=mat.array[r2][j];
          mat.array[r2][j]=temp;
    }
       for (int j=1;j<=b.numCols;j++) {
          temp=b.array[r1][j];
          b.array[r1][j]=b.array[r2][j];
          b.array[r2][j]=temp;
    }
  }


    /**
     * Produce, laddove possibile, la matrice inversa utilizzando l'eliminazione
di Gauss su AX=I mod m
     * dove I è una matrice identità
     * @return una matrice di tipo ModSquareMatrix
     * */
    public ModSquareMatrix inverse() throws MatricesNonConformableException,
SingularMatrixException {
       //Utilizza la classe ModIdentityMatrix
       return gaussianSolve(new ModIdentityMatrix(numRows,modulus));
  }
}
```

- *ModIdentityMatrix.java* : quale sottoclasse della precedente. Tale struttura restituirà la matrice identità necessaria nel calcolo della matrice inversa calcolata nel metodo *inverse()* della classe *ModSquareMatrix.java*.

```
package matrici;


import java.math.BigInteger;

/**
 *ModIdentityMatrix eredita i metodi da ModSquareMatrix e da ModMatrix
 * */
public class ModIdentityMatrix extends ModSquareMatrix {

       /**
        * restituisce una matrice identità
        * @param n di tipo n, rappresenta la dimensione
        * @param mod di tipo BigInteger rappresenta il modulo
        * */
       public ModIdentityMatrix(int n,BigInteger mod) throws
MatricesNonConformableException {
        //invoca il supercostruttore creando una matrice nulla
        super(n,mod,true,false);
        //Setta gli elmenti della diagonale principale ad 1
        for (int i=1;i<=n;i++) array[i][i]=new BigInteger("1");
       }

}
```

Per vedere all'azione queste ultime due classi, riprendiamo i dati del sistema citato in precedenza:

$$A=\begin{pmatrix}1 & 4\\ 2 & 2\end{pmatrix} \qquad X=\begin{pmatrix}x\\ y\end{pmatrix} \qquad B=\begin{pmatrix}3\\ 2\end{pmatrix}$$

e calcoliamone, laddove possibile, le soluzioni del sistema $AX \equiv B \ (mod\ 5)$:

1. calcoliamo $det(A) = (1 \cdot 2)-(4 \cdot 2)=2-8 = -6 \neq 0$, quindi la matrice è invertibile.

   Si noti che in questo caso abbia potuto applicare il *metodo delle diagonali*, poiché la matrice è di ordine 2×2, ma come ci saremmo dovuti comportare se ad esempio la matrice dei coefficienti era di ordine 7×7 ? La risposta la troviamo nell'applicazione dell'eliminazione di Gauss, implementata nel metodo *gaussianSolve* della classe *ModSquareMatrix*, mediante la quale si calcola il determinante mediante il prodotto degli elementi della matrice diagonale ottenuta.

2. Associamo alla matrice A il vettore B dei termini noti:

$$\begin{pmatrix}1 & 4 & 3\\ 2 & 2 & 2\end{pmatrix}$$

3. Sottraiamo per due volte la prima riga alla seconda, e riduciamo sempre il risultato modulo modulo

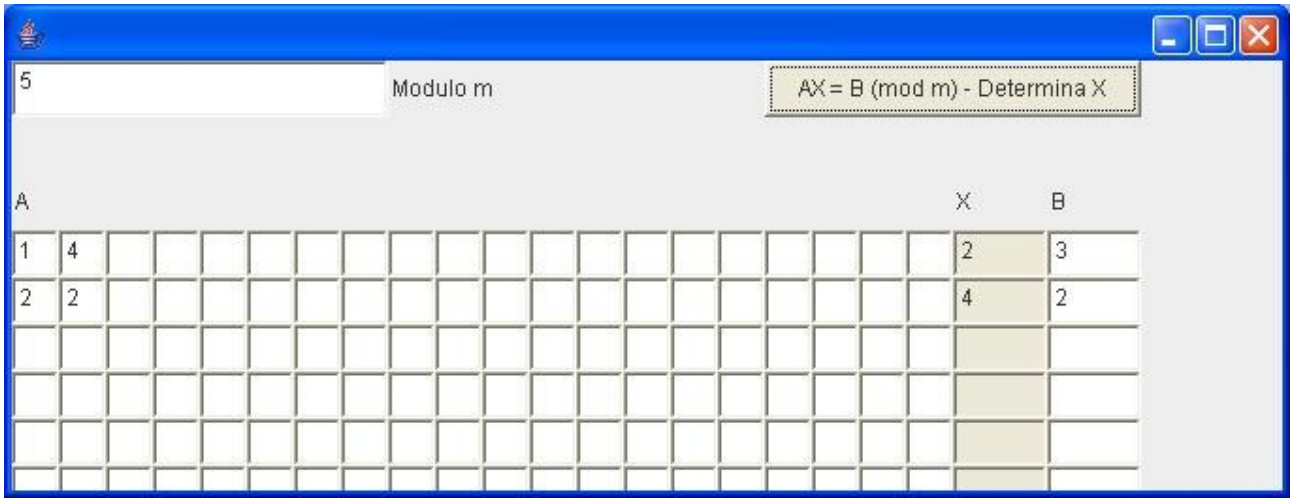
$$\begin{pmatrix}1 & 4 & 3\\ 1 & 3 & 4\end{pmatrix} \rightarrow \begin{pmatrix}1 & 4 & 3\\ 0 & 4 & 1\end{pmatrix}$$

4. dal prodotto della diagonale della matrice dei coefficienti, ricaviamo che il $det(A) \neq 0$, quindi la matrice è invertibile.

5. Procediamo nella moltiplicazione per 4 degli elementi della seconda riga:

$$\begin{pmatrix}1 & 4 & 3\\ 0 & 4 & 1\end{pmatrix} \rightarrow \begin{pmatrix}1 & 4 & 3\\ 0 & 1 & 4\end{pmatrix}$$

6. Con il metodo della *Back-Substitution*, otteniamo le soluzioni del sistema $x=2$ e $y=4$. Tale risultato viene confermato anche dall'esecuzione della classe *TestLinearSystemSolve.java*.




```
package matrici;

import java.awt.*;
import java.awt.event.*;
```

```
import java.math.*;
import javax.swing.*;

public class TestLinearSystemSolve extends JFrame implements ActionListener {

   Label modulusLabel=new Label("Modulo m");
   TextField modulusField=new TextField();
   Button generate=new Button("AX = B (mod m) - Determina X");
   Label errorMsg=new Label("");
   Panel top=new Panel();
   Label lab1=new Label("A");
   Label lab2=new Label("X");
   Label lab3=new Label("B");

   final int maxGridRows=20;
   BigInteger modulus;
   BigInteger[] array;

   MatrixPanel matrixpa=new MatrixPanel(maxGridRows,maxGridRows);
   MatrixPanel matrixpx=new MatrixPanel(maxGridRows,1);
   MatrixPanel matrixpb=new MatrixPanel(maxGridRows,1);

   ModSquareMatrix A;
   ModMatrix X;
   ModMatrix B;

   public TestLinearSystemSolve() {
      setLayout(null);

      top.setSize(600,30);
      top.setLocation(0,0);
      add(top);
      top.setLayout(new GridLayout(1,3));
      top.add(modulusField);
      top.add(modulusLabel);
      top.add(generate);
      generate.addActionListener(this);

      errorMsg.setSize(600,30);
      errorMsg.setLocation(0,30);
      add(errorMsg);

      lab1.setSize(500,30);
      lab1.setLocation(0,60);
      add(lab1);
      lab2.setSize(50,30);
      lab2.setLocation(500,60);
      add(lab2);
      lab3.setSize(50,30);
      lab3.setLocation(550,60);
      add(lab3);

      matrixpa.setSize(500,510);
      matrixpa.setLocation(0,90);
      add(matrixpa);

      matrixpx.setSize(50,510);
      matrixpx.setLocation(500,90);
      add(matrixpx);
      matrixpx.setEditable(false);
```

```
      matrixpb.setSize(50,510);
      matrixpb.setLocation(550,90);
      add(matrixpb);

   pack();
       show();

  }

   public void actionPerformed(ActionEvent e) {
   Object source=e.getSource();
      if (source==generate) {
         try {
            errorMsg.setText("");
            modulus=new BigInteger(modulusField.getText());
            A=readSquareMatrix(matrixpa,maxGridRows);
            B=readVector(matrixpb,maxGridRows);
            X=A.gaussianSolve(B);
            for (int i=0;i<X.rows();i++)
               matrixpx.setEntry(i,0,X.getElement(i+1,1).toString());
         } catch (Exception exc) {
            errorMsg.setText(exc.toString());
    }
   }
  }

   private ModSquareMatrix readSquareMatrix(MatrixPanel mp,int limit) {
      array=new BigInteger[limit*limit];
      int rows=0, columns=0, rowCount=0, columnCount=0;
      String value="";
      while (rowCount<limit&&!(value=matrixpa.getEntry(rowCount,0)).equals(""))
{
     columnCount=0;
         while (columnCount<limit&&!
(value=mp.getEntry(rowCount,columnCount)).equals("")) {
            array[rowCount*columns+columnCount]=new BigInteger(value);
      columnCount++;
     }
         if (rowCount>0&&columnCount!=columns) throw new
IllegalArgumentException("Invalid row/column length");
         else columns=columnCount;
    rowCount++;
   }
   rows=rowCount;
      if (rows==columns) return new ModSquareMatrix(rows,array,modulus);
      else
        throw new IllegalArgumentException("Matrie A non quadrata");
  }

   private ModMatrix readVector(MatrixPanel mp,int limit) {
            array=new BigInteger[limit];
            String value="";
            int r=0;
            while (r<limit&&!(value=mp.getEntry(r,0)).equals("")) {
               array[r]=new BigInteger(value);
      r++;
      }
            return new ModMatrix(r,1,array,modulus);
```

```
  }

    public static void main(String[] arg){
            new TestLinearSystemSolve();
       }//main

}
```

Si noti che i metodi proposti nella classe *ModSquareMatrix.java* si possono applicare anche per la risoluzione, laddove possibile, dell'inversa A' tale che $AA' \equiv A'A \equiv I \ (mod\ 5)$:

1. Associamo alla matrice A la matrice identità I :

$$\begin{pmatrix} 1 & 4 & 1 & 0 \\ 2 & 2 & 1 & 0 \end{pmatrix}$$

2. Sottraiamo per due volte la prima riga alla seconda, e riduciamo sempre il risultato modulo modulo 5

$$\begin{pmatrix} 1 & 4 & 1 & 0 \\ 0 & 4 & 3 & 1 \end{pmatrix}$$

3. Sostituiamo la prima riga con la differenza tra quest'ultimi e la seconda riga :

$$\begin{pmatrix} 1 & 0 & 3 & 4 \\ 0 & 4 & 3 & 1 \end{pmatrix}$$

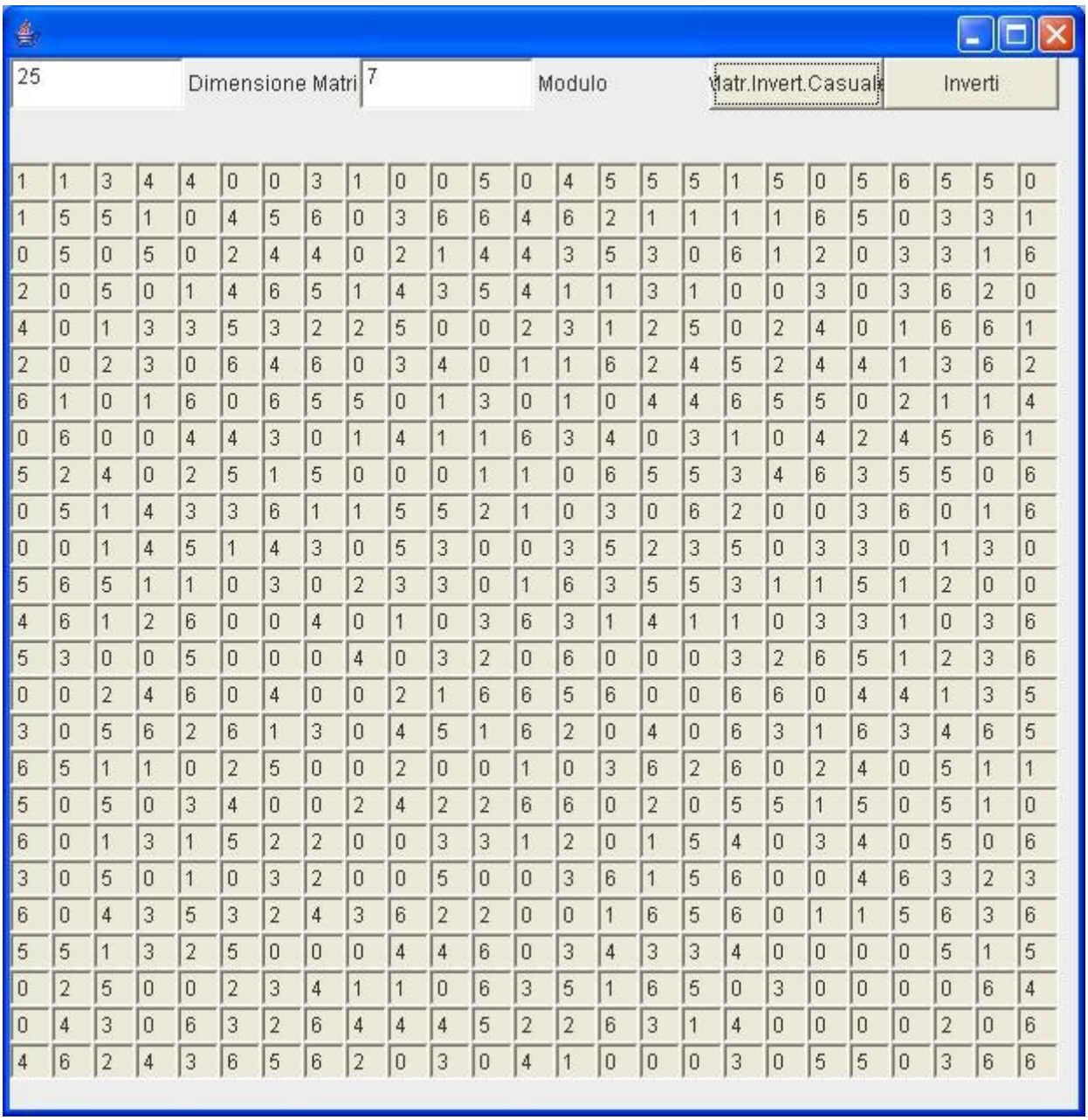
4. moltiplichiamo la seconda riga per 4

$$\begin{pmatrix} 1 & 0 & 3 & 4 \\ 0 & 1 & 2 & 4 \end{pmatrix}$$

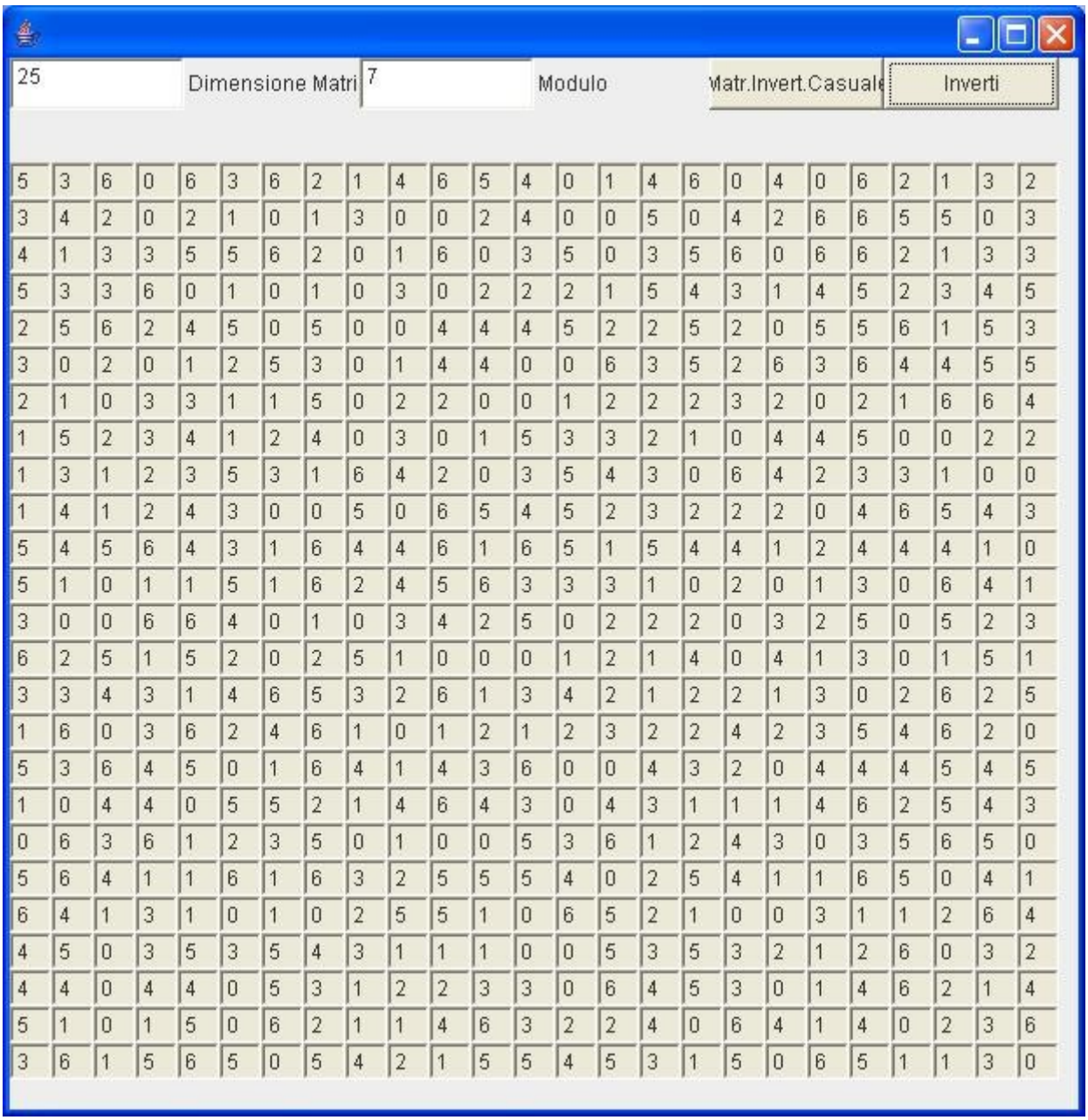
5. per verificare che la matrice ottenuta sia l'inversa, verifichiamo con il prodotto $AA' \equiv I \ (mod\ 5)$:

$$\begin{pmatrix} 1 & 4 \\ 2 & 2 \end{pmatrix} \cdot \begin{pmatrix} 3 & 4 \\ 2 & 4 \end{pmatrix} \equiv \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} mod\ 5$$

A titolo esemplificativo della potenza di calcolo fornita dal metodo *inverte()* della classe *ModSquareMatrix.java*, è stata implementata la classe *TestRandomModSquareMatrix.java* di cui sotto vengono forniti i risultati ottenuti su una matrice di ordine *25* modulo *7*:



1	1	3	4	4	0	0	3	1	0	0	5	0	4	5	5	5	1	5	0	5	6	5	5	0
1	5	5	1	0	4	5	6	0	3	6	6	4	6	2	1	1	1	1	6	5	0	3	3	1
0	5	0	5	0	2	4	4	0	2	1	4	4	3	5	3	0	6	1	2	0	3	3	1	6
2	0	5	0	1	4	6	5	1	4	3	5	4	1	1	3	1	0	0	3	0	3	6	2	0
4	0	1	3	3	5	3	2	2	5	0	0	2	3	1	2	5	0	2	4	0	1	6	6	1
2	0	2	3	0	6	4	6	0	3	4	0	1	1	6	2	4	5	2	4	4	1	3	6	2
6	1	0	1	6	0	6	5	5	0	1	3	0	1	0	4	4	6	5	5	0	2	1	1	4
0	6	0	0	4	4	3	0	1	4	1	1	6	3	4	0	3	1	0	4	2	4	5	6	1
5	2	4	0	2	5	1	5	0	0	0	1	1	0	6	5	5	3	4	6	3	5	5	0	6
0	5	1	4	3	3	6	1	1	5	5	2	1	0	3	0	6	2	0	0	3	6	0	1	6
0	0	1	4	5	1	4	3	0	5	3	0	0	3	5	2	3	5	0	3	3	0	1	3	0
5	6	5	1	1	0	3	0	2	3	3	0	1	6	3	5	5	3	1	1	5	1	2	0	0
4	6	1	2	6	0	0	4	0	1	0	3	6	3	1	4	1	1	0	3	3	1	0	3	6
5	3	0	0	5	0	0	0	4	0	3	2	0	6	0	0	0	3	2	6	5	1	2	3	6
0	0	2	4	6	0	4	0	0	2	1	6	6	5	6	0	0	6	6	0	4	4	1	3	5
3	0	5	6	2	6	1	3	0	4	5	1	6	2	0	4	0	6	3	1	6	3	4	6	5
6	5	1	1	0	2	5	0	0	2	0	0	1	0	3	6	2	6	0	2	4	0	5	1	1
5	0	5	0	3	4	0	0	2	4	2	2	6	6	0	2	0	5	5	1	5	0	5	1	0
6	0	1	3	1	5	2	2	0	0	3	3	1	2	0	1	5	4	0	3	4	0	5	0	6
3	0	5	0	1	0	3	2	0	0	5	0	0	3	6	1	5	6	0	0	4	6	3	2	3
6	0	4	3	5	3	2	4	3	6	2	2	0	0	1	6	5	6	0	1	1	5	6	3	6
5	5	1	3	2	5	0	0	0	4	4	6	0	3	4	3	3	4	0	0	0	0	5	1	5
0	2	5	0	0	2	3	4	1	1	0	6	3	5	1	6	5	0	3	0	0	0	0	6	4
0	4	3	0	6	3	2	6	4	4	4	5	2	2	6	3	1	4	0	0	0	0	2	0	6
4	6	2	4	3	6	5	6	2	0	3	0	4	1	0	0	0	3	0	5	5	0	3	6	6



5	3	6	0	6	3	6	2	1	4	6	5	4	0	1	4	6	0	4	0	6	2	1	3	2
3	4	2	0	2	1	0	1	3	0	0	2	4	0	0	5	0	4	2	6	6	5	5	0	3
4	1	3	3	5	5	6	2	0	1	6	0	3	5	0	3	5	6	0	6	6	2	1	3	3
5	3	3	6	0	1	0	1	0	3	0	2	2	2	1	5	4	3	1	4	5	2	3	4	5
2	5	6	2	4	5	0	5	0	0	4	4	4	5	2	2	5	2	0	5	5	6	1	5	3
3	0	2	0	1	2	5	3	0	1	4	4	0	0	6	3	5	2	6	3	6	4	4	5	5
2	1	0	3	3	1	1	5	0	2	2	0	0	1	2	2	2	3	2	0	2	1	6	6	4
1	5	2	3	4	1	2	4	0	3	0	1	5	3	3	2	1	0	4	4	5	0	0	2	2
1	3	1	2	3	5	3	1	6	4	2	0	3	5	4	3	0	6	4	2	3	3	1	0	0
1	4	1	2	4	3	0	0	5	0	6	5	4	5	2	3	2	2	2	0	4	6	5	4	3
5	4	5	6	4	3	1	6	4	4	6	1	6	5	1	5	4	4	1	2	4	4	4	1	0
5	1	0	1	1	5	1	6	2	4	5	6	3	3	3	1	0	2	0	1	3	0	6	4	1
3	0	0	6	6	4	0	1	0	3	4	2	5	0	2	2	2	0	3	2	5	0	5	2	3
6	2	5	1	5	2	0	2	5	1	0	0	0	1	2	1	4	0	4	1	3	0	1	5	1
3	3	4	3	1	4	6	5	3	2	6	1	3	4	2	1	2	2	1	3	0	2	6	2	5
1	6	0	3	6	2	4	6	1	0	1	2	1	2	3	2	2	4	2	3	5	4	6	2	0
5	3	6	4	5	0	1	6	4	1	4	3	6	0	0	4	3	2	0	4	4	4	5	4	5
1	0	4	4	0	5	5	2	1	4	6	4	3	0	4	3	1	1	1	4	6	2	5	4	3
0	6	3	6	1	2	3	5	0	1	0	0	5	3	6	1	2	4	3	0	3	5	6	5	0
5	6	4	1	1	6	1	6	3	2	5	5	5	4	0	2	5	4	1	1	6	5	0	4	1
6	4	1	3	1	0	1	0	2	5	5	1	0	6	5	2	1	0	0	3	1	1	2	6	4
4	5	0	3	5	3	5	4	3	1	1	1	0	0	5	3	5	3	2	1	2	6	0	3	2
4	4	0	4	4	0	5	3	1	2	2	3	3	0	6	4	5	3	0	1	4	6	2	1	4
5	1	0	1	5	0	6	2	1	1	4	6	3	2	2	4	0	6	4	1	4	0	2	3	6
3	6	1	5	6	5	0	5	4	2	1	5	5	4	5	3	1	5	0	6	5	1	1	3	0

Teorema Cinese dei resti

Nel paragrafo precedente sono stati considerati sistemi lineari di congruenze, tutte eventi lo stesso modulo. In questo paragrafo ci accingiamo ad affrontare la risoluzione di sistemi lineari di congruenze con moduli differenti. A tal proposito consideriamo il seguente sistema di congruenze:

$x \equiv c_1 \ (mod\ r_1)$

$x \equiv c_2 \ (mod\ r_2)$

...

$x \equiv c_s \ (mod\ r_s)$

con $MCD(r_i,r_j)=1 \ \forall(i,j) \wedge i \neq j$, ammette una soluzione. Tale soluzione è unica modulo $R = r_1 r_2 ... r_s$.

Consideriamo di dover contare rapidamente un gruppo di studenti di una scuola, ma non di vere il tempo di chiamarli uno per uno, l'unica informazione che abbia a disposizione è che sono meno di

1000. Come possiamo procedere? Possiamo chiedergli di mettersi in file per *5*, per *8* e per *19* e ogni volta che si sono allineati, ne contiamo i ragazzi che restano fuori. Tali resti c_i per $i=1,2,3$ saranno rispettivamente minori di *5*, 8 *e 19*. Fatto ciò, risolviamo il seguente sistema di congruenze:

$x \equiv c_1 \pmod 5$

$x \equiv c_2 \pmod 8$

$x \equiv c_3 \pmod{19}$

Applicando il Teorema Cinese dei Resti, sappiamo che il sistema ammette un'unica soluzione modulo $R=5\times8\times19=760$, la quale indicherà il numero degli studenti.

Supponiamo che quando si allineano per *5* ne resto fuori *1*, quando si allineano per *8* ne restino fuori *2* e quando si allineano per *19* ne restino fuori *3*, allora porremo $c_1 = 1$, $c_2 = 2$, $c_3 = 3$ tale che:

$x \equiv 1 \pmod 5$

$x \equiv 2 \pmod 8$

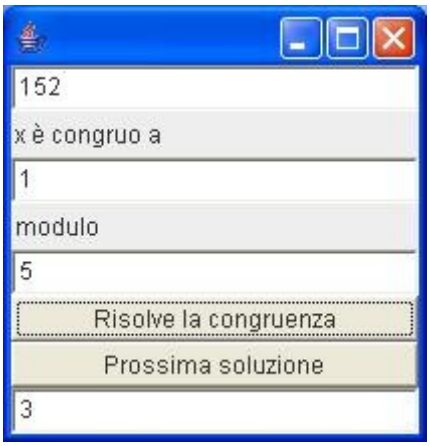
$x \equiv 3 \pmod{19}$

quindi calcoliamo i coefficienti R_i :

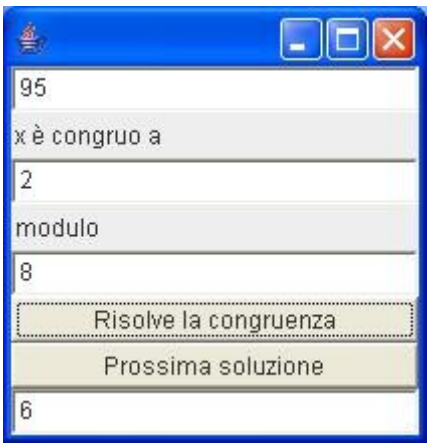
- $R_1 = r_2 \times r_3 = 8 \times 19 = 152$
- $R_2 = r_1 \times r_3 = 5 \times 19 = 95$
- $R_3 = r_1 \times r2 = 5 \times 8 = 40$

risolviamo le seguenti equazioni di congruenze, sfruttando il metodo *solveLinearCongruence* della classe *BigIntegerMath.java*:

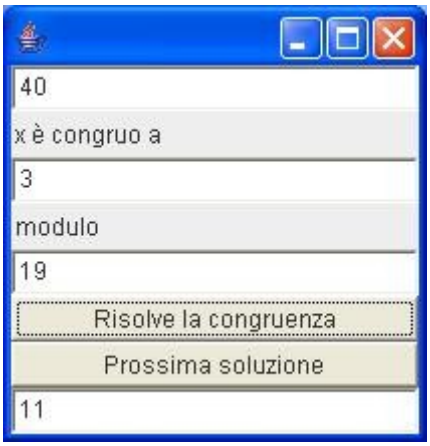
- $R_1x_1 \equiv 1 \pmod 5 \rightarrow 152x_1 \equiv 1 \pmod 5$



- $R_2x_2 \equiv 2 \pmod 8 \rightarrow 95x_2 \equiv 2 \pmod 8$



- $R_3x_3 \equiv 3 \pmod{19} \rightarrow 40x_3 \equiv 3 \pmod{19}$



e quindi la soluzione unica del sistema è data da:

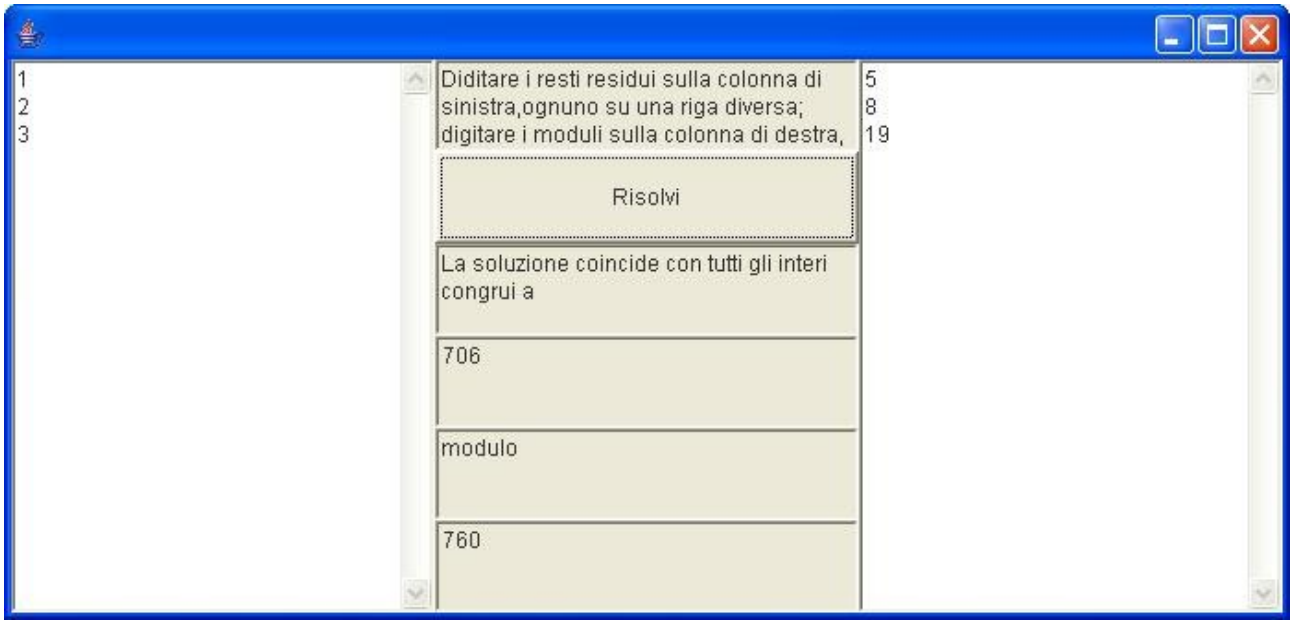
$$x_0 = R_1x_1 + R_2x_2 + R_3x_3 = (152\times\ 3) + (95\times\ 6) + (40\times\ 11)\ = 1466 \equiv 706 \pmod{760}$$

Per risolvere tale tipo di problematica si è provveduto ad inserire nella classe *BigIntegerMath.java* il metodo *solveCRT*:

```
/**
	* Risolve un sistema lineare di congruenze x = b(i) (mod m(i) )
	* utilizzando il teorema cinese dei resti
	* @param residue di tipo BigInteger[], rappresenta il vettore dei resti
residui
	* @param modulus di tipo BigInteger[], rappresenta il vettore dei moduli
	* @exception se i resti ed i moduli non corrispondono o se i moduli non
sono a due a due coprimi
	* @return answer[] di tipo BigInteger, tale che answer[0] conterrà le
soluzioni residue,
	* mentre answer[1] conterrà il prodotto di ciascun modulo
	* */
      public static BigInteger[] solveCRT(BigInteger[] residue, BigInteger[]
modulus) {
            //Verifica se i resti residui ed i moduli coincidano
            if (residue.length!=modulus.length)
                  throw new IllegalArgumentException("Resti e moduli non
corrispondono.");
            //Verifica se i moduli sono a due a due coprimi
            for (int i=0; i<modulus.length-1; i++) {
                  for (int j=i+1; j<modulus.length; j++) {
                        if (!(modulus[i].gcd(modulus[j]).equals(ONE)))
                              throw new IllegalArgumentException("I moduli non
```

```
sono coprimi.");
                  }
            }
            //effettua il prodotto di ogni modulo
            BigInteger M = new BigInteger("1");
            for (int i=0; i<modulus.length; i++)
                  M=M.multiply(modulus[i]);
            //formula la soluzione come nel teorema Cinese dei resti
            BigInteger solution=new BigInteger("0");
            for (int i=0;i<modulus.length; i++) {
                  BigInteger Mi=M.divide(modulus[i]);


solution=solution.add(residue[i].multiply(Mi).multiply(Mi.modInverse(modulus[i])
));
            }
            solution=lnr(solution,M);

            BigInteger[] result=new BigInteger[2];
            result[0]=solution;
            result[1]=M;

            return result;

      }//solveCRT
```

Il quale viene sfruttato nell'interfaccia grafica definita nella classe *TestCRT.java*



Congruenze polinomiali non lineari

Consideriamo una congruenza del tipo

$$f(x) \equiv 0 \pmod{m}$$

dove *f(x)* è un polinomio in *x* a coefficienti interi del quale cerchiamo soluzioni modulo un intero positivo *m* arbitrario.

Nei paragrafi precedenti abbiamo studiato le congruenze lineari, ossia le congruenze polinomiali di grado *1*. Ora faremo alcune considerazioni sulle congruenze polinomiali non lineari, soffermandoci dal punto di vista implementativo sulle congruenze di grado *2*.

Partiamo dal caso in cui m sia un numero primo p. É immediata conseguenza del teorema di Ruffini che un polinomio a coefficienti in un campo non può avere più radici del suo grado. Allora la congruenza $f(x) \equiv 0 \pmod p$, dove p è primo, non potrà avere più di n soluzioni, se n è il grado di $f(x) \in Z_p[x]$. Ciò non è vero rispetto ad un modulo non primo. Infatti si consideri la seguente congruenza di grado 2 e modulo 8:

$$x^2-1 \equiv 0 \pmod 8$$

ammette 4 soluzioni, cioè $x=1,3,5$ e 7.

In generale se $m = p_1^{\alpha 1} \times p_2^{\alpha 2} \times \ldots \times p_r^{\alpha r}$ ovvero è fattorizzabile come prodotto di primi distinti, allora la congruenza polinomiale $f(x) \equiv 0 \pmod p$ è risolubile se e solo se è risolubile ciascuna congruenza $f(x) \equiv 0 \pmod{p_i^{\alpha i}} \ \forall i=1,2,3,\ldots,k$.

Equazioni di secondo grado

Consideriamo l'equazione di congruenza *quadratica* espressa nella seguente forma:

$$ax^2+ bx + c \equiv 0 \pmod m$$

tale che m=p, ove p è un primo dispari maggiore di 2 e non divide il coefficiente a. Tale tipo di equazione, la si può risolvere con l'uso del *complemento* quadratico.

Sia $a' \in Z$ tale che $2aa' \equiv 1 \pmod p$. Un tale valore a' esiste sempre poiché $2a \neq 0$ in Z_p. Moltiplicando $ax^2+ bx + c \equiv 0 \pmod p$ per $2a'$ otteniamo la seguente equazione di congruenza equivalente:

$2a(ax^2+ bx + c) \equiv 0 \pmod p \rightarrow 2a'ax^2+ 2a'bx + 2a'c) \equiv 0 \pmod p \mid 2aa' = 1,\ a'b = b' \text{ e } 2a'c = c' \rightarrow$

$x^2+ 2b'x + c' \equiv 0 \pmod p \leftrightarrow (x+ b)^2 \equiv b'^2 - c' \pmod p$

ponendo $y = x+b'$ e $k = b'^2 - c'$, l'equazione quadratica si riduce alla forma

$$y^2 \equiv k \pmod p$$

Per risolvere la $ax^2+ bx + c \equiv 0 \pmod p$ basta risolvere $x+ b' \equiv y \pmod p$ *dove* y è la soluzione di $y^2 \equiv k \pmod p$. Si noti che la congruenza $x+ b' \equiv y \pmod p$ ammette sempre un'unica soluzione mentre l'equazione $y^2 \equiv k \pmod p$ può non avere alcuna soluzione, ovvero avere due soluzioni, una opposta all'altra. Tali due soluzioni possono essere coincidenti in una con molteplicità 2 se $k \equiv 0 \pmod p$.

Quindi sulla base di tali premesse, consideriamo i seguenti due esempi:

- Si vogliono trovare le soluzioni della congruenza quadratica

  $$2x^2+ x + 3 \equiv 0 \pmod 5$$

  determiniamo in primis il termine $a' \in Z$ tale che $2aa' \equiv 1 \pmod p$ ovvero $22a' \equiv 1 \pmod 5$ che è uguale a $4a' \equiv 1 \pmod 5$ la cui soluzione è $a' = 4 \equiv -1 \pmod 5$, quindi $2a'=2(-1)=-2$, il quale moltiplicherà la congruenza quadratica proposta:

  $-2(2x^2+ x + 3) \equiv 0 \pmod 5 \rightarrow -4x^2- 2x - 6 \equiv 0 \pmod 5$ la quale è congrua modulo 5 a $x^2- 2x -1 \equiv 0 \pmod 5$ ed equivalente a $(x+b')^2 \equiv b'^2 - c' \pmod p$ ovvero $(x-1)^2 \equiv 2 \pmod 5$.

  Ponendo $y=x-1$, la congruenza da risolvere è pertanto la

  $y^2 \equiv 2 \pmod 5 \rightarrow y^2- 2 \equiv 0 \pmod 5$

  le cui soluzioni si ricavano sostituendo i valori $y=0,1,2,3,4$ o ancor meglio dal punto di vista computazionale $y=0,1,2,-1,-2$ tale che:

- $f(0) = -2 \equiv 3 \pmod 5$
- $f(1) = -1 \equiv 4 \pmod 5$
- $f(2) = 2$
- $f(-1) = f(1)$
- $f(-2) = f(2)$

dai cui risultati, si deduce che la congruenza associata $y^2 - 2 \equiv 0 \pmod 5$ non ammette soluzioni, quindi neanche la congruenza originaria $2x^2 + x + 3 \equiv 0 \pmod 5$

- Si vogliono trovare le soluzioni della congruenza quadratica

$$3x^2 + x + 1 \equiv 0 \pmod 5$$

determiniamo in primis il termine $a' \in Z$ tale che $2aa' \equiv 1 \pmod p$ ovvero $23a' \equiv 1 \pmod 5$ che è uguale a $6a' \equiv 1 \pmod 5$ la cui soluzione è $a' = 6 \equiv 1 \pmod 5$, quindi $2a' = 2(1) = 2$, il quale moltiplicherà la congruenza quadratica proposta:

$2(3x^2 + x + 1) \equiv 0 \pmod 5 \rightarrow 2(1)3x^2 - 2(1)(1)x + 2(1)1 \equiv 0 \pmod 5$ in cui $b' = (1)1 = 1$ e $c' = 2(1)1 = 2 \rightarrow 6x^2 - 2x + 2 \equiv 0 \pmod 5$ la quale è congrua modulo 5 a $x^2 + 2x + 2 \equiv 0 \pmod 5$ ed equivalente a $(x+b')^2 \equiv b'^2 - c' \pmod p$ ovvero $(x+1)^2 \equiv -1 \pmod 5$ cioè $(x+1)^2 \equiv 4 \pmod 5$.

Ponendo $y = x+1$, la congruenza da risolvere è pertanto la

$y^2 \equiv 4 \pmod 5 \rightarrow y^2 - 4 \equiv 0 \pmod 5$

le cui soluzioni si ricavano sostituendo i valori $y = 0,1,2,3,4$ o ancor meglio dal punto di vista computazionale $y = 0,1,2,-1,-2$ tale che:

- $f(0) = -4 \equiv 1 \pmod 5$
- $f(1) = -3 \equiv 2 \pmod 5$
- $f(2) = 0$
- $f(-1) = f(1)$
- $f(-2) = f(2) = 0$

dai cui risultati, si deduce che la congruenza associata $y^2 - 4 \equiv 0 \pmod 5$ ammette soluzioni per $y_1 = 2$ e $y_2 = -2 \equiv 3 \pmod 5$, quindi la congruenza originaria $3x^2 + x + 1 \equiv 0 \pmod 5$ ammette le seguenti soluzioni ricavate da $y = x+1 \rightarrow x = y-1$:

- $x_1 = y_1 - 1 = 2 - 1 = 1 \pmod 5$
- $x_2 = y_2 - 1 = 3 - 1 = 2 \pmod 5$

Per risolvere tale tipo di problematiche, è stato implementato il metodo statico *solveQuadratic()* all'interno della classe *BigIntegerMath.java*.

Per la corretta esecuzione di tale metodo, vine ipotizzato che il modulo sia prodotto di primi distinti p e q che soddisfino la forma $4k+3$, ovvero che sia congruo $3 \bmod(4)$.

Nel caso in cui questi due fattori rappresentino grandi interi, bisogna prendere in considerazione i seguenti accorgimenti:

- forniamo il modulo già fattorizzato in due primi distinti p e q (il problema della fattorizzazione lo tratteremo in seguito).

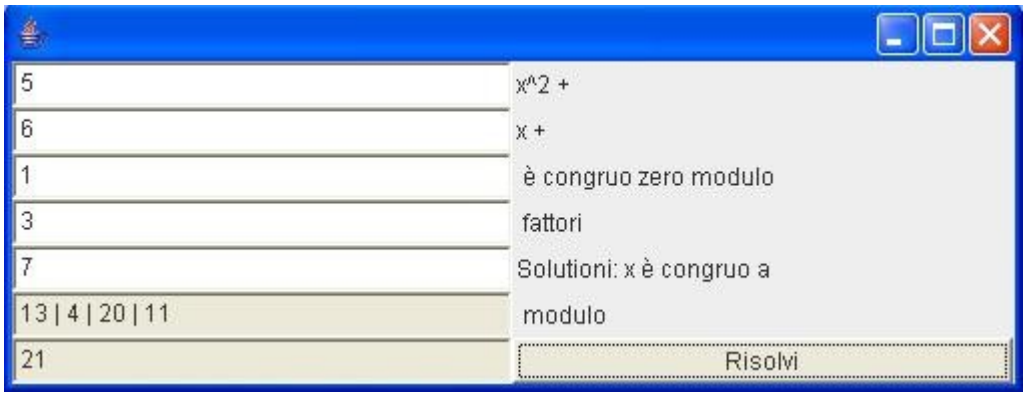
- Bisogna testare gli aspiranti fattori primi *p* e *q* mediante il metodo probabilistico *isProbablePrime()* della classe *BigInteger* . Se questi risulta *true*, bisogna verificare se il fattore *p* o *q* sia congruo *3 mod(4)*. Se il test probabilistico ci restituisce *false*, il metodo solleverà un'eccezione.
- Bisogna accettarsi che i fattori del modulo *n* non si ripetano, ovvero che i due primi *p* e *q* siano unici. In tale sede utilizzeremo il metodo CRT, per l'applicazione del Teorema Cinese dei Resti, il quale richieste che i moduli siano a due a due coprimi tra loro, quindi nel caso in cui i fattori *p* e *q* siano uguali o comunque non coprimi tra loro, verrà sollevata un'eccezione.
- Bisogna controllare le soluzioni che otteniamo. Infatti, è possibile che la congruenza quadratica che stiamo cercando di risolvere, non abbia soluzioni! Quindi, ogni valore che otteniamo va sottoposto alla congruenza originaria. Se ogni soluzione proposta, non risolve quest'ultimi, verrà sollevata un'eccezione.

Come visto il metodo proposto ha molte regole da seguire, ma, se prendiamo delle buone precauzioni, questi può produrre risultati molto soddisfacenti. Ecco il codice:

```
/**
 * Risolve equazione di congruenze quadratiche del tipo ax^2+bx+c congruo 0 mod
n=pq
 * @param a di tipo BigInteger, rappresenta il coefficiente della variabile x^2
 * @param b di tipo BigInteger, rappresenta il coefficiente della variabile x
 * @param c di tipo BigInteger, rappresenta il termine noto
 * @param p di tipo BigInteger, rappresenta il il primo fattore primo di n
 * @param q di tipo BigInteger, rappresenta il il secondo fattore primo di n
 * @param primeTolerance di tipo int, rappresenta
 * @exception Se il test probabilistico isProbablePrime() ci restituisce false,
il metodo solleverà un'eccezione
 * @exception IllegalArgumentException nel caso in cui i fattori p e q siano
uguali o comunque non coprimi tra loro, verrà sollevata un'eccezione
 * @exception IllegalArgumentException nel caso in cui i fattori non siano
congrui 3 modulo 4
 * @exception Se ogni soluzione proposta, non risolve la congruenza originaria,
verrà sollevata un'eccezione.
 * @return un array di 4 BigInteger, laddove le soluzioni esistono
 * */
public static BigInteger[] solveQuadratic(BigInteger a, BigInteger b, BigInteger
c,    BigInteger p, BigInteger q, int primeTolerance) {
      //verifica se i fattori del modulo sono differenti
      if (p.equals(q))
            throw new IllegalArgumentException("I fattori del modulo non sono
unici!");
      //Verifica che i fattori siano congrui 3 modulo 4
      BigInteger n=p.multiply(q);
      if (!lnr(p.mod(FOUR),n).equals(THREE))
            throw new IllegalArgumentException(p+" non è congruo 3 modulo 4!");
      if (!lnr(q.mod(FOUR),n).equals(THREE))
            throw new IllegalArgumentException(q+" non è congruo 3 modulo 4!");
      //Verifica se i fattori del modulo siano primi
      if (!p.isProbablePrime(primeTolerance))
            throw new IllegalArgumentException(p+" non è primo!");
      if (!q.isProbablePrime(primeTolerance))
            throw new IllegalArgumentException(q+" non è primo!");
      //Crea l'array delle soluzioni
      BigInteger[] result=new BigInteger[4];
      //Inizia a formare i termini
      BigInteger aInv=a.modInverse(n);
      BigInteger pInv=p.modInverse(q);
```

```
        BigInteger qInv=q.modInverse(p);
        BigInteger twoInv=TWO.modInverse(n);
        BigInteger
term1=aInv.multiply(twoInv.multiply(b).modPow(TWO,n).multiply(aInv).subtract(c))
;
        BigInteger term2=twoInv.multiply(aInv).multiply(b);
        BigInteger
t1=lnr(term1.modPow(p.add(ONE).divide(FOUR),n).subtract(term2).multiply(q).multi
ply(qInv),n);
        BigInteger
t2=lnr(term1.modPow(q.add(ONE).divide(FOUR),n).subtract(term2).multiply(p).multi
ply(pInv),n);
        BigInteger
t3=lnr(term1.modPow(p.add(ONE).divide(FOUR),n).negate().subtract(term2).multiply
(q).multiply(qInv),n);
        BigInteger
t4=lnr(term1.modPow(q.add(ONE).divide(FOUR),n).negate().subtract(term2).multiply
(p).multiply(pInv),n);
        //forma la soluzione
        result[0]=lnr(t1.add(t2),n);
        result[1]=lnr(t1.add(t4),n);
        result[2]=lnr(t3.add(t2),n);
        result[3]=lnr(t3.add(t4),n);
        //Verifica la soluzione, se qualcuno non soddisfa la congruenza originaria
si solleva un'eccezione
        BigInteger x;
        for (int i=0;i<4;i++) {
              x=result[i];
              if (!
lnr(a.multiply(x.multiply(x)).add(b.multiply(x)).add(c),n).equals(ZERO))
                    throw new IllegalArgumentException("Soluzione x="+x+" non
verificata!");
        }

        return result;
}
```

La cui interfaccia grafica definita nella classe *TestSolveQuadratic.java*




```
package algebra;

import java.math.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class TestSolveQuadratic extends JFrame implements ActionListener {
```

```
 BigInteger a;
 BigInteger b;
 BigInteger c;
 BigInteger p;
 BigInteger q;
 BigInteger n;
 BigInteger[] answer;

 TextField af=new TextField(30);
 Label label1=new Label("x^2 + ");
 TextField bf=new TextField(30);
 Label label2=new Label("x + ");
 TextField cf=new TextField(30);
 Label label3=new Label(" è congruo zero modulo ");
 TextField pf=new TextField(30);
 Label label4=new Label(" fattori ");
 TextField qf=new TextField(30);
 Label label5=new Label("Solutioni: x è congruo a ");
 TextField ansArea=new TextField(30);
 Label label6=new Label(" modulo ");
 TextField nf=new TextField(80);
 Button solveButton=new Button("Risolvi");

 public TestSolveQuadratic() {

       Panel p1=new Panel();

       p1.setLayout(new GridLayout(7,2));
       p1.add(af);
       p1.add(label1);
       p1.add(bf);
       p1.add(label2);
       p1.add(cf);
       p1.add(label3);
       p1.add(pf);
       p1.add(label4);
       p1.add(qf);
       p1.add(label5);
       p1.add(ansArea);
       ansArea.setEditable(false);
       p1.add(label6);
       p1.add(nf);
       nf.setEditable(false);
       p1.add(solveButton);
       solveButton.addActionListener(this);
      add(p1);
      pack();
      show();
}

 public void actionPerformed(ActionEvent e) {
   String outString="";
   if (e.getSource()==solveButton) {
     try {
       a=new BigInteger(af.getText());
       b=new BigInteger(bf.getText());
       c=new BigInteger(cf.getText());
       p=new BigInteger(pf.getText());
       q=new BigInteger(qf.getText());
       n=p.multiply(q);
```

```
            ansArea.setText("");
            answer=BigIntegerMath.solveQuadratic(a,b,c,p,q,16);
            for (int i=0;i<answer.length-1;i++) {
               outString+=answer[i]+" | ";
        }
            outString+=answer[answer.length-1].toString();
            ansArea.setText(outString);
            nf.setText(n.toString());
          } catch (NumberFormatException nfe) {
            ansArea.setText(nfe.toString());
          } catch (IllegalArgumentException iae) {
            ansArea.setText(iae.toString());
          } catch (ArithmeticException ae) {
            ansArea.setText(ae.toString());
      }
     }
    }

  public static void main(String[] arg){
            new TestSolveQuadratic();
       }//main

}
```

Metodi di Fattorizzazione

Come già anticipato nel paragrafo precedente, chi aspira ad entrare nel mondo della crittoanalisi, spesso si troverà di fronte ad un problema, che dal punto di vista computazionale risulta essere molto oneroso, ovvero, la fattorizzazione di un intero, sopratutto quando questi risulta essere molto grande, ad esempio dell'ordine di un centinaio di cifre.

Conoscere la fattorizzazione di un intero molto grande in due primi è come essere in possesso di un'*informazione segreta*. Questo è il principio su cui si basa la crittografia a chiave pubblica, la quale verrà presentata nel capitolo successivo: l'informazione pubblica è l'intero, emtre il segreto è la sua fattorizzazione in due primi.

In questo paragrafo affronteremo tre tipi di fattorizzazione:

1. il Crivello di Eratostene;
2. il metodo di fattorizzazione di Fermat;
3. il metodo di fattorizzazione *p-1* di Pollard.

Il Crivello di Eratostene

Uno dei primi metodi che ci apprestiamo a presentare è il ben noto *Crivello di Eratostene*, il quale è un antico procedimento per il calcolo delle tabelle di numeri primi fino ad un certo numero *n* prefissato. Deve il nome al matematico Eratostene di Cirene, che ne fu l'ideatore. Pur non essendo un algoritmo straordinariamente efficiente, infatti, è in compenso piuttosto semplice da tradurre in un qualsiasi linguaggio di programmazione.

Il procedimento è il seguente: si scrivono tutti i naturali a partire da *2* fino n in un elenco detto *setaccio*, il quale verrà implementato da un array. Poi si cancellano, setacciando appunto, tutti i multipli del primo numero del setaccio escluso esso stesso. Si prosegue così fino ad arrivare in fondo. I numeri che restano sono i numeri primi minori od uguali a n. È come se si utilizzassero dei setacci a maglie via via più larghe: il primo lascia passare solo i multipli di 2, il secondo solo i

multipli di 3, e così via.

Nel caso *n = 50*, ad esempio, il procedimento di setacciatura si conclude con il numero *7* perché *7* è il massimo intero il cui quadrato non supera *50* e si può provare che il procedimento di setacciatura per ricercare i primi fino ad un certo numero *n* cessa sempre quando si supera la radice quadrata di *n*.

L'algoritmo appena descritto è stato implementato nella classe *Eratostene.java* , di cui in seguito viene proposto il codice sorgente:

```
package algebra;

/**
 * Crivello di Eratostene
 * calcola tutti i numeri primi fino ad un dato numero,
 * sia questo ad esempio 100;
 * Complessità : esponenziale (da usare solo per numeri piccoli)
 * */
class Eratostene{

	//limite superiore della serie di numeri primi
	static int MAX=50;
	//lista dei numeri
	int [] numeri = new int[MAX];
	//lista dei numeri primi, vale true se il rispettivo indice è primo
	boolean [] primi = new boolean[MAX];

	//costruttore
	public Eratostene(){
		//buffer
		int i;
		//fase di inizializzazione
		for (i=0;i<MAX;i++){
			numeri[i]=i+1;
			primi[i]=true;
		};
	}//Eratostene

	/**
	 * @param a di tipo int, numero di cui si determinano
	 * i multipli
	 * Elimina i multipli di a
	 * */
	void elMul(int a){
		for (int j=2;j*a<=MAX;j++)
			primi[(j*a)-1]=false;
	}

	/**
	 * @param a di tipo int, numero primo di cui si
	 * vuole ottenere il successivo
	 * Determina l'elemento primo successivo
	 * */
	int prossimo(int n){
		int tmp=n;
		while (!primi[tmp]) {
			tmp++;
			if (tmp>=MAX) break;
		}//while
		return tmp+1;
	}//prossimo
```

```
	/**
	 * determina i numeri primi
	 */
	void calcolaPrimi(){
		int num=2;
		while (num<=MAX){
			elMul(num);
			num=prossimo(num);
		};
	} //calcolaPrimi

	/**
	 * Visualizza a video la successione di numeri primi
	 * minori di MAX
	 * */
	void scriviPrimi() {
		System.out.println("I numeri primi fino a "+MAX+" sono:");
		for (int i=0; i < MAX ; i++){
			if (primi[i])
				System.out.print(numeri[i]+" ");
		}//for

	} //scriviPrimi

	public static void main(String[] arg){
		Eratostene e = new Eratostene();
		e.calcolaPrimi();
		e.scriviPrimi();
	}//main

}//Eratostene
```

il quale fornisce fornisce il seguente output:

```
I numeri primi fino a 50 sono:
1 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

Il metodo di fattorizzazione di Fermat

Nel paragrafo precedente, abbiamo visto come funziona il Crivello di Eratostene per fattorizzare un numero n, in cui occorre vedere se esso è divisibile per i numeri minori od uguali a radice quadrata di n. Abbiamo anche notato tuttavia che questo metodo è in generale assai dispendioso dal punto di vista della complessità del calcolo. Talvolta è più efficiente il seguente metodo dovuto a Fermat:

1. si può supporre n dispari;
2. nel caso in cui n sia dispari fattorizzarlo equivale a determinare due interi x e y tali che:

   $$n = x^2 - y^2$$

   Infatti, se $n = x^2 - y^2$, allora $n = (x+y)(x-y)$ è una fattorizzazione di n. Viceversa, se $n = ab$, allora, supposto $a \geq b \geq 1$, si può scrivere:

   $$n = \left(\frac{a+b}{2}\right)^2 + \left(\frac{a-b}{2}\right)^2$$

   dove $(a+b)/2$ e $(a-b)/2$ sono interi non negativi. Infatti, essendo n dispari, anche a e b sono dispari e quindi $a \pm b$ è pari;
3. determinare x e y tali che $n = x^2 - y^2$ equivale a determinare x tale che $y^2 = x^2 - n$.

4. si determina il più piccolo intero positivo k tale che $k^2 \geq n$
5. si calcolano la differenze $k^2 - n$, $(k + 1)^2 - n$,$(k + 2)^2 - n$, ... fino a che si trova un valore $t \geq k$ tale che t^2-n sia un quadrato.

   Si noti che tale ciclo terminerà al più tardi quando $t = (n+1) / 2$, perché

$$\left(\frac{n+1}{2}\right)^2 - n = \left(\frac{n-1}{2}\right)^2$$

   valore che si ottiene quando il numero n è primo e quindi ammette solo la fattorizzazione banale:

$$n = \left(\frac{n+1}{2} + \frac{n-1}{2}\right)^2 \cdot \left(\frac{n+1}{2} + \frac{n-1}{2}\right)^2 = n \cdot 1$$

Consideriamo il seguente esempio, con il quale si vuole fattorizzare il numero *29591* in fattori primi con appunto il metodo di Fermat, ponente $k = 173$:

- $173^2 - 29591 = 338$ che non è un quadrato
- $174^2 - 29591 = 685$ che non è un quadrato
- $175^2 - 29591 = 1034$ che non è un quadrato
- $176^2 - 29591 = 1385$ che non è un quadrato
- $177^2 - 29591 = 1738$ che non è un quadrato
- $178^2 - 29591 = 2093$ che non è un quadrato
- $179^2 - 29591 = 2450$ che non è un quadrato
- $180^2 - 29591 = 2809 = 53^2$ è un quadrato

quindi ricaviamo che $29591 = (180 + 53)(180 - 53) = 233 \times 127$

Il metodo appena presentato è stato generalizzato da Kraitchick al modo seguente: anziché ricercare due interi x e y tali che $n = x^2 - y^2$ si richiede di cercare due *interi* x e y tali che

$$x^2 - y^2 \equiv 0 \pmod{n}$$

Ovviamente una coppia di interi x e y che soddisfa tale congruenza non garantisce la fattorizzazione di n: dice solamente che n divide $x^2 - y^2 = (x+y)(x-y)$.

Naturalmente può succedere che *tutti* i fattori primi di n dividano uno solo dei due numeri x-y, x+y: in questo caso si avrebbe *che* $x \equiv y \pmod{n}$ oppure $x \equiv -y \pmod{n}$.

Se tuttavia se non è verificata la congruenza $x \equiv \pm y \pmod{n}$, per trovare un fattore *non banale* di n basta calcolare il $MCD(n, x-y)$ o $MCD(n, x+y)$.

A tal proposito, consideriamo il seguente esempio, con il quale si vuole fattorizzare il numero *559* in fattori primi con la generalizzazione del metodo di Fermat.

Poniamo di conoscere già la coppia $(x,y) = (41,2)$ avremo che $41^2 \equiv 2^2 \pmod{559}$. Determinano allora $MCD(n , x-y) = MCD(559 , 41-2) = MCD(559 , 39) = 13$, il quale è un fattore primo di *559*. La fattorizzazione di *559* è quindi data da $559 = 13 \times (559/13) = 13 \times 43$.

Per sviluppare un metodo che estragga i fattori del numero n usando il Metodo di Fermat, c'è bisogno dell'implementazione di un metodo che determini la radice quadrata di n. La classe *BigInteger*, la quale fino ad ora abbiamo utilizzato come struttura dati di default per le nostre implementazioni, non fornisce un metodo ad hoc, quindi abbiamo provveduto in merito.

L'algoritmo che ci apprestiamo a presentare, restituisce il maggior numero intero il cui quadrato non superi *n*. Per calcolare quindi la radice quadrata di un intero positivo, utilizzeremo un algoritmo numerico noto come *metodo di Newton*.

Supponiamo che vogliamo trovare la radice quadrata di *n*, ovvero, una soluzione di

$$x^2=n$$

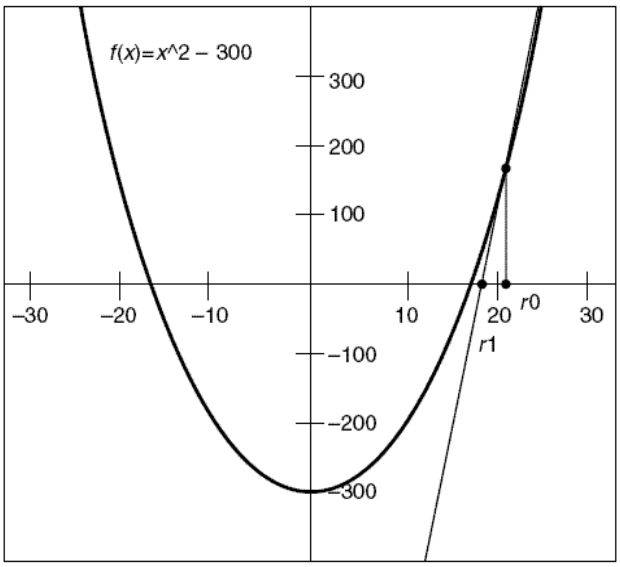
è possibile farlo, se siamo in grado di determinare una radice (o zero) della funzione

$$f(x) = x^2 - n$$

Per utilizzare il metodo di Newton, dobbiamo determinare la derivata della *f(x)* :

$$f'(x) = 2x$$

Scegliamo un valore iniziale r_0, tale che questi sia una sovrastima della radice cercata.



A partire da tale valore calcoliamo la seguente sequenza:

$$r_k=\frac{r_{k-1}-f(r_{k-1})}{f'(r_{k-1})}\forall\, k=1,2,3,\ldots$$

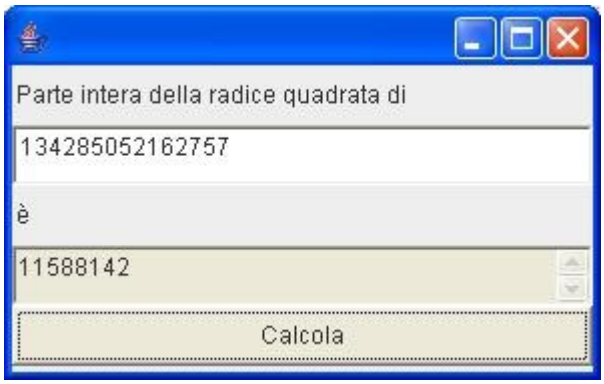
Tale approccio, restituisce una radice di *f(x)* in tempi rapidi. Se la radice che stiamo cercando è un irrazionale, troncheremo il risultato in modo tale da ottenere un valore intero.

A tal proposito la classe *BigIntegerMath.java*, è stata dotata del metodo *sqrt()* il quale determina il valore della radice quadrata di un intero positivo *n* mediante il Metodo di Newton appena illustrato.

```
/**
 * Determina il maggior numero intero il cui quadrato
 * non superi <i>m.</i>
 * Essenzilamente cerchiamo la radice della funzione
 * f(x)=x^2-p t.c. f'(x)=2x
 * @param m di tipo BigInteger, rappresenta il valore per cui si
 * vuole determinare il maggior numero intero il cui quadrato
 * non superi appunto <i>m.</i>
 * @exception Solleva un accezione nel caso in cui viene fornito un parametro
negativo
 * @exception Solleva un accezione nel caso in cui viene fornito un parametro
nullo
 * */
public static BigInteger sqrt(BigInteger m) {
```

```
        int diff=m.compareTo(ZERO);
        //Solleva un accezione nel caso in cui viene fornito un parametro negativo
        if (diff<0)
              throw new IllegalArgumentException("Cannot compute square root of a
negative integer!");
        //Restituisce 0 nel caso in cui m sia nullo
        if (diff==0)
              return BigInteger.valueOf(0);
        BigDecimal two=new BigDecimal(TWO);
        //Converte il parametro in BigDecimal
        BigDecimal n=new BigDecimal(m);
        //Setta il valore iniziale pari alla metà del valore di m
        byte[] barray=new byte[m.bitLength()/16+1];
        barray[0]=(byte)255;
        BigDecimal r=new BigDecimal(new BigInteger(1,barray));
        //le successive approssimazioni verranno definite da r-f(r)/f'(r)
r=r.subtract(r.multiply(r).subtract(n).divide(r.multiply(two),BigDecimal.ROUND_U
P));
        //Continueremo ad approssimare fino a quando non si eccede rispetto al
valore stesso di m
        while (r.multiply(r).compareTo(n)>0) {
r=r.subtract(r.multiply(r).subtract(n).divide(r.multiply(two),BigDecimal.ROUND_U
P));
        }//while

        //Tronca le parti decimali
        return r.toBigInteger();
}//sqrt
```

Per testare questo metodo è stata implementa la classe *TestSQRT.java*, .




```
package algebra;

import java.math.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class TestSQRT extends JFrame implements ActionListener {

   Label l1=new Label("Parte intera della radice quadrata di ");
   TextField numField=new TextField();
   Label l2=new Label("è");
   TextArea sqrtArea=new TextArea();
   Button doit=new Button("Calcola");

   public TestSQRT() {
```

```
        Panel p1=new Panel();

        p1.setLayout(new GridLayout(5,1));
        p1.add(l1);
        p1.add(numField);
        p1.add(l2);
        p1.add(sqrtArea);
        sqrtArea.setEditable(false);
        p1.add(doit);
        doit.addActionListener(this);
        add(p1);
        pack();
        show();
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource()==doit) {
            try {
                BigInteger n=new BigInteger(numField.getText());
                sqrtArea.setText(BigIntegerMath.sqrt(n).toString());
            } catch (Exception exc) {
                sqrtArea.setText(exc.toString());
            }
        }
    }

    public static void main(String[] arg){
            new TestSQRT();
        }//main
}
```

mentre, la fattorizzazione con il metodo di Fermat è stata implementa con il metodo *fermat* della classe *BigIntegerMath.java*

```
/**
 * Implementa il Metodo di Fattorizzazione di fermat
 * Sia <i>n</i> un intero positivo, e
 * <i>a*b</i> una fattorizzazione dell'intero <i>n</i>
 * con due interi positivi.
 * Allora possiamo scrivere il numeri <i>n</i>
 * come prodotto della differnza tra due radici.
 * <i> n = a*b = s^2 - t^2 </i>
 * dove <i> s = (a+b)/2 </i>
 * e <i> t = (a-b)/2 </i>
 * @param n di tipo BigInteger, rappresenta il numero <i>n</i> da fattorizzare
 * @exception Il numero intero deve essere maggiore di tre
 * @return un array di due elementi contenenti i due fattori
 **/
public static BigInteger[] fermat(BigInteger n)throws IllegalArgumentException {

      // Variabili
      boolean solved = false;
      BigInteger i;
      // Limite superiore
      BigInteger L;
      BigInteger r;
      BigInteger t = ONE;
      BigInteger s = ONE;
      //fattori
      BigInteger[] factors = new BigInteger[2];
```

```
        if (n.compareTo(THREE)<=0)
                throw new IllegalArgumentException("Il numero intero deve essere
maggiore di tre!");

        // Determina la radice quadrata
        r = sqrt(n);
        //Bisogna aver fortuna a finché r sia un quadrato perfetto
        if( (r.multiply(r)).compareTo(n) == 0) {
                solved = true;
                factors[0] = r;
                factors[1] = r;
        }// Se r^2 < n
        else if((r.multiply(r)).compareTo(n) < 0) {
                //Assicurarsi che r^2 sia più grande di n
                while( (r.multiply(r)).compareTo(n) < 0)
                        r = r.add(ONE);
        }//else if

        //Memorizza il valore di partenza
        i = r;
        //Imposta il limite superiore a (n+1)/2 - sqrt(n)
        L = ((n.add(ONE)).divide(TWO)).subtract(r);
        //Avvia la soluzione
        while(!solved && r.compareTo(L) < 0) {
                t = r.multiply(r).subtract(n);
                s = sqrt(t);
                if(s.multiply(s).compareTo(t) !=0)
                        r = r.add(ONE);
                else {
                        t = r.subtract(s);
                        s = r.add(s);
                        solved = true;
                        factors[0] = t;
                        factors[1] = s;
                }
        }

        // Nessun fattore
        if(!solved){
                factors[0] = NO;
                factors[1] = NO;
        }

        return factors;

}//fermat
```

il quale è stato testato con la classe *TestFermat.java*

```
package algebra;

import java.math.BigInteger;
import algebra.BigIntegerMath;

/**
 * Classe di test per il metodo fermat della classe
 * BigIntegerMath
 **/
public class TestFermat {
```

```
	public static final BigInteger NO=new BigInteger("-1");
	public static final BigInteger ZERO=new BigInteger("0");
	public static final BigInteger ONE=new BigInteger("1");
	public static final BigInteger TWO=new BigInteger("2");
	public static final BigInteger THREE=new BigInteger("3");
	public static final BigInteger FOUR=new BigInteger("4");

	public static void main( String[] args) {

		// Variabili
		int bitlength = 25;
		//Dimensione dei numeri primi

		//Imposta il primo fattore
		BigInteger p1 = new BigInteger( "16" );
		//Imposta il secondo fattore
		BigInteger p2 = new BigInteger( "16" );
		// Il nostro numero da scomporre
		BigInteger n = new BigInteger( "16" );

		//Genera, a caso, un numero dispari composito
		while(n.mod(TWO).compareTo(ONE) !=0) {

			//genera un valore da un test probabilistico per p2 da 32 bit
			p1 = BigIntegerMath.getPrimeNbit(bitlength);
			if(p1.isProbablePrime(100))
				System.out.print("Primo p1: " );
			System.out.println( p1 +" lungo " + p1.bitLength() +" bit");


			//genera un valore da un test probabilistico per p2 da 32 bit
			p2 = BigIntegerMath.getPrimeNbit(bitlength);
			if(p2.isProbablePrime(100))
				System.out.print("Primo p2: " );
			System.out.println( p2 +" lungo " + p2.bitLength() +" bit");

			n = new BigInteger("5157437");
			n = p1.multiply(p2);
		}
		System.out.print("Numero primo composto n: ");
		System.out.println( n +" lungo " + n.bitLength() +" bit");

		BigInteger[] factors = BigIntegerMath.fermat(n);

		if(factors[0].compareTo(NO)==0)
			System.out.println("Impossibile determinare una
fattorizzazione !!");
		else
			System.out.print(n + " = " +factors[0] +" * " +factors[1]);

	}

}
```

il cui output risulta essere il seguente:

```
Primo p1: 30058283 lungo 25 bit
```

Primo p2: 24001207 lungo 25 bit
Numero primo composto n: 721435072347581 lungo 50 bit
```
721435072347581 = 24001207 * 30058283
```

Il metodo di fattorizzazione ρ di Pollard

Il metodo presentato nel paragrafo precedente è l'algoritmo che sta alla base di tecniche di fattorizzazione molto più complesse, le quali vanno adottate solo per numeri interi di grandi cifre, alcuni tra questi sono:

- il metodo delle basi di fattorizzazione;
- il metodo del crivello quadratico.

Il metodo di fattorizzazione che ora introduciamo è basato su un'idea del tutto diversa da quella del metodo di fattorizzazione di Fermat.

Supponiamo di sapere che un numero n è composto. Per decomporlo in fattori primi procediamo, lavorando su Z_n, nel seguente modo:

1. Sia n l'intero da fattorizzare e Z_n l'anello[4] su cui è definita l'applicazione

   $$f: Z_n \to Z_n$$

   tale che f sia facilmente calcolabile, come ad esempio un polinomio $f(x)$ a coefficienti interi che non sia lineare e non dia luogo ad un'applicazione biunivoca : una buona scelta più delle volte è il polinomio $f(x) = x^2 + 1$.
2. Fissiamo un elemento arbitrario $x_o \in Z_n$.
3. Creiamo una successione di interi a partire da x_o tale che
   - $f(x_o)=x_1$;
   - $f(x_1)=f^2(x_o)=x_2$;
   - ...
   - $f^i(x_o)=f(x_{i-1})=x_i$;

4 Ricordiamo che un anello è una struttura algebrica composta da un insieme A su cui sono definite due operazioni binarie, chiamate somma e prodotto e indicate rispettivamente con + e •, che godono di proprietà simili a quelle verificate dai numeri interi ovvero:

(A, +) è un gruppo abeliano con elemento neutro 0:

* (a + b) + c = a + (b + c)
* a + b = b + a
* 0 + a = a + 0 = a
* ∀a ∃(−a) tale che a + −a = −a + a = 0

(A, ·) è un semigruppo:

* (a·b)·c = a·(b·c)

La moltiplicazione è distributiva rispetto alla somma:

* a·(b + c) = (a·b) + (a·c)
* (a + b)·c = (a·c) + (b·c)
* 1·a = a·1 = a

(le relazioni devono valere per ogni a, b e c in A)

tutto ovviamente *modulo n*.

4. Confrontiamo i risultati $x_k - x_j \forall\ k > j$ tali che non valga la congruenza $x_k \equiv x_j\ (mod\ n)$, ovvero interi distinti in Z_n , ma bensì valga la congruenza $x_k \equiv x_j\ (mod\ d)$ tale che $d|n$.
5. Una volta definita la coppia $(k\ ,\ j)$, calcoliamo $MCD(\ x_k - x_j\ ,\ n)$ il quale sarà un divisore proprio di n.

Consideriamo ad esempio il caso in cui si voglia fattorizzare il numero $n=111$ utilizzando la funzione $f(x) = x^2 + 1$ definita in Z_{299} con $x_o=1$, allora procederemo nella seguente maniera:

1. calcoliamo i valori della successione:
   - $x_o = 1$
   - $x_1 = f^1(x_o) = 1^2+1 = 2$
   - $x_2 = f^2(x_1) = 2^2+1 = 5$
   - $x_3 = f^2(x_2) = 5^2+1 = 26$
   - $x_4 = f^2(x_3) = 26^2+1 = 677 \equiv 11\ (mod\ 111)$
   - $x_5 = f^2(x_4) = 11^2+1 = 122 \equiv 11\ (mod\ 111)$

   avendo ritrovato dunque la stessa classe, quindi ci fermiamo.
2. Effettuiamo le differenze e calcoliamo $MCD(\ x_k - x_j\ ,\ 111)$:
   - $x_1 - x_0 = 2 - 1 = 1$ e $MCD(1\ ,111) = 1$
   - $x_2 - x_0 = 5 - 1 = 4$ e $MCD(4\ ,111) = 1$
   - $x_3 - x_0 = 26 - 1 = 25$ e $MCD(25\ ,111) = 1$
   - $x_4 - x_0 = 11 - 1 = 10$ e $MCD(10\ ,111) = 1$
   - $\mathbf{x_2 - x_1 = 5 - 2 = 3}$ **e** $\mathbf{MCD(3\ ,111) = 3}$

   abbiamo trovato un divisore proprio di 111 e possiamo dire che $111 = 3 \times 37$.

Risulta ovvio che per numeri così piccoli non è certo il caso di scomodare questi algoritmi; è ben chiaro che 111 è divisibile per 3.

Quando il numero è grande, calcolare *tutti* i $MCD(\ x_k - x_j\ ,\ 111)$, per ogni $j < i$, può essere molto oneroso. Il problema verte essenzialmente nel minimizzare il numero di calcoli *del MCD* per ogni coppia $(k\ ,\ j)$, quindi con la seguente revisione del metodo appena esposto, ci proponiamo di calcolare il *MCD* per ogni x_k solo su un x_j.

Nell'esempio precedente abbiamo determinato che $x_2 \equiv x_1\ (mod\ 111)$, quindi impostiamo l'indice $k = 2$ e compiliamo la seguente tabella, fermandoci non appena identifichiamo un fattore primo di n:

$\mathbf{k_{10}}$	$\mathbf{k_2}$	$\mathbf{h = length(k_2)-1}$	$\mathbf{j = 2^h-1}$	$\mathbf{x_k - x_j}$	$\mathbf{MCD(x_k - x_j, 111)}$
1	1	0	$2^0-1 = 0$	$x_1 - x_0 = 2 - 1 = 1$	MCD (1, 111) = 1
2	01	1	$2^1-1 = 1$	$x_2 - x_1 = 5 - 2 = 3$	MCD(3 , 111) = 3

Come si può vedere abbiamo identificato un fattore primo in soli due passaggi. Tale miglioria è stata realizzata *PollardRho* della classe *BigIntegerMath.java*.

```
/**
 * Fattorizzazione Rho di Pollard
 * Sia <i>n</i> un grande numero intero composto, e sia <i>d</i> il suo
 * divisore più piccolo.
 * Fissiamo una successione di interi x[0],x[1],...x[s] tali che sono
caratterizzati
 * dall'avere minimi resti residui non negativi  mod p non tutti distinti i cui
 * valori sono definiti con f^i(x[0])=f (x[i-1] )=x[i], ove f(x) è una funzione
```

```
 * biunivoca del tipo biunivoca : una buona scelta più delle volte è il
polinomio
 * f(x) = x^2 + 1.
 * Confrontiamo i risultati x[k] - x[j] per ogni k > j tali che non valga la
 * congruenza  x[k] congruo ?x[j] (mod n), ovvero interi distinti in Z[n] , ma
bensì
 * valga la congruenza x[k] congruo ?x[j] (mod d) tale che d|n
 * Una volta definita la coppia (k , j), calcoliamo MCD( x[k] - x[j] , n),
tramite
 * l'algoritmo Euclideo, il quale sarà un divisore proprio di n.
 * Quindi avremo che n = MCD( x[k] - x[j] , n)*(n/MCD( x[k] - x[j] , n))
 * @param n di tipo BigInteger, rappresenta il numero da fattorizzare
 * @exception il parametro <i>n</i> deve essere maggiore di tre
 * @return un valore di tipo BigInteger rappresentante il fattore di <i>n</i>,
questi varrà -1 se banale
 **/
public static BigInteger PollardRho(BigInteger n)throws IllegalArgumentException
{

	SecureRandom random = new SecureRandom();
	// Variabili
	boolean solved = false;
	// n-1
	BigInteger n1;
	//Dimensione del numero primo
	int bitlength = 32;
	// Limite superiore
	BigInteger L = TWO;
	// resto
	BigInteger d = ONE;
	//primo contatore
	BigInteger i = ONE;
	// secondo contatore
	BigInteger k = TWO;
	//seconda parte
	BigInteger x1;
	BigInteger y;
	BigInteger temp;

	if (n.compareTo(THREE)<=0)
		throw new IllegalArgumentException("Il numero intero deve essere
maggiore di tre!");

	// Sia x1 uguale ad un numero casuale compreso tra 0 e n-1
	n1 = n.subtract(ONE);
	temp = n1;
	// Imposta la lunghezza di n1 in termini di bit
	bitlength = n1.bitLength();
	// temp >= n-1
	while (temp.compareTo(n1) >=0) {
		temp = new BigInteger(bitlength, random);
	}
	x1 = temp;
	//y = x[i]
	y = x1;
	// Imposta un valore massimo pari alla radice quadrata di n
	L = sqrt(n);
	// Determina i valori della successione mediante il polinomio f(x) = x^2+1
```

```
	while (!solved && i.compareTo(L) < 0) {
		// i = i + 1
		i = i.add(ONE);
		//x[i] = (x[i-1]^2+1) mod n
		x1 = ((x1.pow(2)).add(ONE)).mod(n);
		// d = MCD(y - x[i], n)
		d = gcd(y.subtract(x1), n);

		// if ( d != 1 and d != n)
		if (d.compareTo(ONE) != 0 && d.compareTo(n) !=0) {
			solved = true;
		}//if

		// if ( i == k)
		if ( i.compareTo(k) == 0) {
			// y = x[i]
			y = x1;
			// k = k*2
			k = k.multiply(TWO);
		}//if

	}//while

	// Nessun fattore
	if(!solved)
		return NO;
	else //fattore
		return d;

}//PollardRho
```

e testato con la classe *TestPollardRho.java*

```
package algebra;

import java.math.BigInteger;

/**
 * Classe di test per il metodo PollardRho della
 * classe BigIntegerMath.java
 **/

public class TestPollardRho {

	public static final BigInteger NO=new BigInteger("-1");
	public static final BigInteger ONE=new BigInteger("1");
	public static final BigInteger TWO=new BigInteger("2");
	public static final BigInteger THREE=new BigInteger("3");

	public static void main( String[] args) {

		// Variabili

		//Dimensione del numero primo
		int bitlength = 32;
		// primo fattore primo
		BigInteger p1;
		// secondo fattore primo
		BigInteger p2;
		// n = p*q
```

```
            BigInteger n = new BigInteger("5157437");
            // resto
            BigInteger d;

            //genera un valore da un test probabilistico per p2 da 32 bit
            p1 = BigIntegerMath.getPrimeNbit(bitlength);
            if(p1.isProbablePrime(100))
                  System.out.print("Primo p1: " );
            System.out.println( p1 +" lungo " + p1.bitLength() +" bit");


            //genera un valore da un test probabilistico per p2 da 32 bit
            p2 = BigIntegerMath.getPrimeNbit(bitlength);

            if(p2.isProbablePrime(100))
                  System.out.print("Primo p2: " );
            System.out.println( p2 +" lungo " + p2.bitLength() +" bit");
            // Crea un numero composto
            n = p1.multiply(p2);
            System.out.print("Numero Composto n = p1*p2 : " );
            System.out.println( n +" lungo " + n.bitLength() +" bit");

            d = BigIntegerMath.PollardRho(n);

            // if ( d != 1 and d != n)
            if (d.compareTo(ONE) != 0 && d.compareTo(n) !=0) {
                  System.out.print( n +" = " + d + " * " +n.divide(d));
            }else
                  System.out.println("Impossibile trovare una
fattorizzazione !");

      }//main

}//TestPollardRho
```

il cui output risulta essere il seguente:

```
Primo p1: 2278861001 lungo 32 bit
```
Primo p2: 3880442977 lungo 32 bit
Numero Composto n = p1*p2 : 8842990166889639977 lungo 63 bit
```
8842990166889639977 = 2278861001 * 3880442977
```

Il metodo di fattorizzazione *p-1* di Pollard

Supponiamo di dover fattorizzare un intero n. Il metodo che presentiamo funziona nel caso in cui n possiede un fattore primo p, da determinare tale che $p-1$ non possiede divisori primi troppo grandi.

Prima di procedere all'illustrazione dell'algoritmo in questione, vale la pena ricordare due tematiche fondamentali le quali risulteranno utili per l'argomentazione di tale metodo di fattorizzazione, ovvero:

- Calcolo delle potenze modulo n.
- Piccolo Teorema di Fermat.

Calcolo delle potenze modulo n

Dato un intero a e due interi m ed n ci proponiamo di calcolare

$$a^m \ (mod\ n)$$

Possiamo supporre che sia $a < n$ e che ogni volta che facciamo una moltiplicazione, contestualmente riduciamo il risultato modulo n; comunque gli interi che si incontreranno non saranno mai maggiori di n^2.

Scriviamo l'esponente m in base 2 tale che $m_{(2)} = n_0 + 2n_1 + 2^2n_2 + ... + 2^{k-1}n_{k-1}$ con $n_j \in \{0,1\}$ e per cui

$$a^m = a^{n_0+2n_1+2^2n_2+...+2^{k-1}n_{k-1}} = a^{n_0}\left(a^2\right)^{n_1}...\left(a^{2^{k-1}}\right)^{n_{k-1}}$$

all'inizio del procedimento poniamo $p_0=1$. Effettueremo $k-1$ passi, tali che $k-1$ è la lunghezza di m in base 2, definendo di volta in volta $p_1,p_2,...,p_{k-1},p_k$,ed alla fine troveremo che $p_k = a^m \pmod n$.

- se $n_0 = 1$, allora poniamo $p_1=ap_0=a1 =a$, cioè poniamo $p_1=ap_0 \pmod n$, altrimenti lasciamo $p_1=p_0=1$;
- successivamente eleviamo a al quadrato e poniamo $a_1 = a^2 \pmod n$. Se $n_1 = 1$, allora poniamo $p_2=a_1p_1 \pmod n$, altrimenti lasciamo $p_2=p_1$;
- successivamente eleviamo a_1 al quadrato e poniamo $a_2 = a_1^2 \pmod n$. Se $n_2 = 1$, allora poniamo $p_3=a_2p_2 \pmod n$, altrimenti lasciamo $p_3=p_2$;

continuando in questo maniera, al passo j-esimo calcoliamo $a_j = a_{j-1}^{2^j} \pmod n$. Se $n_j = 1$, ossia se 2^j compare nell'esposizione binaria di m, allora poniamo $p_{j+1}=a_jp_j \pmod n$, altrimenti lasciamo $p_{j+1}=p_j$;

Dopo $k-1$ passi si ottiene così $p_k = a^m \pmod n$.

Proviamo ora a calcolare $3^{13} \pmod 7$ con l'algoritmo esposto: in questo caso abbiamo che $m=13$ quindi $m_{(2)} = 1101 = 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$ tale che $n_3 = 1$, $n_2 = 1$, $n_1 = 0$, $n_0 = 1$, e la sua lunghezza è data data $|m_{(2)}|-1 = k-1 = 4-1 = 3$. Quindi effettueremo $k-1 = 3$ passi, definendo p_0,p_1,p_2,p_3,p_4 ,tale che $p_4 = a^m \pmod n$, inoltre si noti che $a=a_0=3$ e poniamo $p_0=1$.

- Dato che $n_0 = 1$, allora poniamo $p_1=a_0p_0=3\times1 =3$;
- successivamente calcoliamo $a_0^2=3^2=9$ e poniamo $a_1 = a_0^2 \pmod n = 9 \pmod 7 = 2$. Essendo $n_1 = 0$, poniamo $p_2=p_1=3$;
- successivamente calcoliamo $a_1^2=2^2=4$ e poniamo $a_2 = a_1^2 \pmod n = 4 \pmod 7 = 4$. Essendo $n_2 = 1$, poniamo $p_3=a_2p_2 \pmod n = 4\times3 = 12 \equiv 5 \pmod 7$;
- successivamente calcoliamo $a_2^2=4^2=16\equiv 2 \pmod 7$ e poniamo $a_3 = a_2^2 \pmod n = 2 \pmod 7 = 2$. Essendo $n_3 = 1$, poniamo $p_4=a_3p_3 \pmod n = 2\times5 = 10 \equiv 3 \pmod 7$;

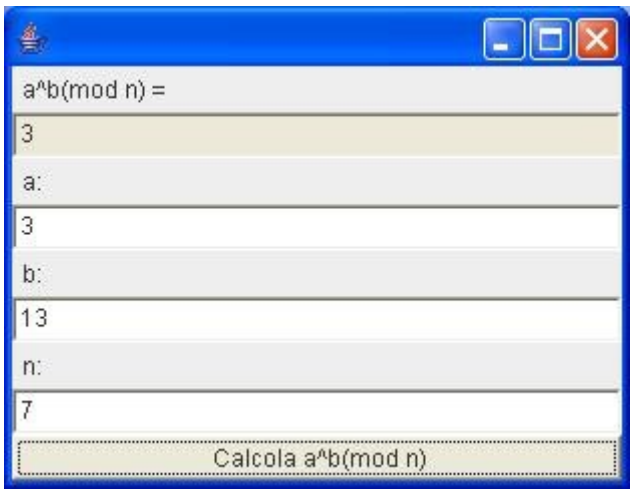
quindi il risultato di $3^{13} \pmod 7 = p_4 = 3$.

L'algoritmo appena esposto è stato implementato e testato nella classe *TestModPow.java*

```
package algebra;

import java.math.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class TestModPow extends JFrame implements ActionListener {

   Label gcdLabel=new Label(" a^b(mod n) = ");
   TextField msg=new TextField(40);
   Label num1Label=new Label(" a: ");
   TextField entryField1=new TextField(40);
   Label num2Label=new Label(" b: ");
   TextField entryField2=new TextField(40);
   Label modulusLabel=new Label(" n: ");
```

```
    TextField modulus=new TextField(40);
    Button gebModPow=new Button(" Calcola a^b(mod n)");

    public TestModPow() {

          Panel p1=new Panel();

          p1.setLayout(new GridLayout(9,1));
          p1.add(gcdLabel);
          p1.add(msg);
          msg.setEditable(false);
          p1.add(num1Label);
          p1.add(entryField1);
          p1.add(num2Label);
          p1.add(entryField2);
          p1.add(modulusLabel);
          p1.add(modulus);
          p1.add(gebModPow);
          gebModPow.addActionListener(this);
          add(p1);
          pack();
          show();
    }

    public void actionPerformed(ActionEvent e) {
       if (e.getSource()==gebModPow) {
          //BigInteger g=null;
          try {
             BigInteger a=new BigInteger(entryField1.getText());
             BigInteger b=new BigInteger(entryField2.getText());
             BigInteger n=new BigInteger(modulus.getText());
             a = a.modPow(b,n);
             msg.setText(a.toString());
          } catch (NumberFormatException nfe) {
             msg.setText("Non è un intero!");
          }
       }
    }

    public static void main(String[] arg){
             new TestModPow();
       }//main

}
```

il cui output risulta essere il seguente:



Piccolo Teorema di Fermat

Sia *a* un intero e *p* un numero primo. Allora

$$a^p \equiv a(mod\ p)$$

e se *a* non è divisibile per *p*, si ha

$$a^{p-1} \equiv 1(mod\ p)$$

Tale teorema si può usare per creare un test di primalità, o, più precisamente, per verificare se un numero *non* è primo, in quanto se esiste *a* tale che non vale la congruenza $a^n \equiv a(mod\ n)$ allora *n* non è primo.

Si noti che questo test di *non primalità*, se funziona, ci assicura che il numero *non è primo*, ma non ci permette di trovare una fattorizzazione di *n*.

Ritorniamo al metodo di fattorizzazione *p-1 di Pollard*: ci occorre trovare una numero *k* che sia diviso da *p-1*. A tale scopo possiamo procedere nel modo seguente: sia $p\text{-}1 = 2^a 3^b 5^c \ldots T^z$ la fattorizzazione in fattori primi di *p-1*. Poiché *p* non è noto, non sono noti neanche gli esponenti *a,b,c,...* tuttavia essendo $2^a \leq p\text{-}1 < n$, allora avremo che

$$a < \frac{\log(n)}{\log(2)}, b < \frac{\log(n)}{\log(3)}, \ldots$$

in definitiva posto

$$k = 2^{\frac{\log(n)}{\log(2)}} \cdot 3^{\frac{\log(n)}{\log(3)}} \ldots T^{\frac{\log(n)}{\log(T)}}$$

si ha che *p-1* divide *k*. Sia *a* un intero compreso tra *2* e *n-2*, tale che *MCD(a,n)=1*, sicché *a* è primo con tutti i fattori primi di *n*. Per il *piccolo teorema di Fermat* si ha che $a^{p-1} \equiv 1(mod\ p)$ e quindi, dato che $p\text{-}1 \mid k$, $a^k \equiv 1(mod\ p)$.

A questo punto:

- calcoliamo $a^k \equiv 1(mod\ n)$ e contestualmente mediante l'algoritmo Euclideo delle divisioni successive, calcoliamo $MCD(a^k\text{-}1\ ,\ n)=d$. Ovviamente il primo *p* che stiamo cercando è tale che $p|d$, quindi $d \neq 1$.
- Se $d \neq n$, abbiamo trovato un fattore, non necessariamente primo, di *n* ed abbiamo finito il procedimento;
- altrimenti se $d = n$, cioè se $n \mid (a^k\text{-}1)$, bisogna cambiare la scelta dell'intero *a* o dell'intero *k* e

ricominciare. In pratica si prende come *k* un multiplo di tutti gli interi minori o uguale di un prefissato *M*, il quale si suppone maggiore di tutte le potenze dei numeri primi che dividono *p-1*. Ad esempio si può considerare *k=M!*

A titolo di esempio fattorizziamo il numero *156203* con il metodo sopra esposto:

- Poniamo *M=6* quindi *k=M!=6!=720* ed *a=2*
- Calcoliamo $a^k \ (mod\ n) = 2^{720} \ (mod\ 156203)=181$
- Abbiamo trovato che *181*, il quale è un numero primo, è un fattore di *156203* e la sua fattorizzazione è data da *156203 / 181 = 863* quindi $156203 = 181 \times 863$

L'algoritmo appena esposto è stato implementato nel metodo *pMinusOneFactor* della classe *BigIntegerMath.java*.

```
/**
 * Implementa il metodo di fattorizzazione p-1 di Pollard
 * @param n di tipo BigInteger, rappresenta il numero da fattorizzare
 * @return un valore di tipo BigInteger rappresentante il fattore di <i>n</i>
 * */
public static BigInteger pMinusOneFactor(BigInteger n)throws
IllegalArgumentException {

	if (n.compareTo(THREE)<=0)
		throw new IllegalArgumentException("Il numero intero deve essere
maggiore di tre!");
	Random rand=new Random();
	BigInteger power=BigInteger.valueOf(1);
	BigInteger residue=lnr(BigInteger.valueOf(rand.nextInt()),n);
	BigInteger test=residue.subtract(ONE);
	BigInteger gcd=test.gcd(n);
	while (true) {
		while (gcd.equals(ONE)) {
			power=power.add(ONE);
			residue=residue.modPow(power,n);
			test=residue.subtract(ONE);
			gcd=test.gcd(n);
		}//while
		if (gcd.equals(n)) {
			power=BigInteger.valueOf(1);
			residue=lnr(BigInteger.valueOf(rand.nextInt()),n);
			test=residue.subtract(ONE);
			gcd=test.gcd(n);
		} else
			return gcd;
	}//while

}//pMinusOneFactor
```

e testato con la classe *TestpMinusOneFactor.java*

```
package algebra;

import java.math.BigInteger;

public class TestpMinusOneFactor {

	public static final BigInteger NO=new BigInteger("-1");
	public static final BigInteger ONE=new BigInteger("1");
	public static final BigInteger TWO=new BigInteger("2");
	public static final BigInteger THREE=new BigInteger("3");
```

```
	public static void main( String[] args) {

		// Variabili

		//Dimensione del numero primo
		int bitlength = 32;
		// primo fattore primo
		BigInteger p1;
		// secondo fattore primo
		BigInteger p2;
		// n = p*q
		BigInteger n = new BigInteger("5157437");
		// resto
		BigInteger d;

		//genera un valore da un test probabilistico per p2 da 32 bit
		p1 = BigIntegerMath.getPrimeNbit(bitlength);
		if(p1.isProbablePrime(100))
			System.out.print("Primo p1: " );
		System.out.println( p1 +" lungo " + p1.bitLength() +" bit");


		//genera un valore da un test probabilistico per p2 da 32 bit
		p2 = BigIntegerMath.getPrimeNbit(bitlength);

		if(p2.isProbablePrime(100))
			System.out.print("Primo p2: " );
		System.out.println( p2 +" lungo " + p2.bitLength() +" bit");
		// Crea un numero composto
		n = p1.multiply(p2);
		System.out.print("Numero Composto n = p1*p2 : " );
		System.out.println( n +" lungo " + n.bitLength() +" bit");

		d = BigIntegerMath.pMinusOneFactor(n);

		// if ( d != 1 and d != n)
		if (d.compareTo(ONE) != 0 && d.compareTo(n) !=0) {
			System.out.print( n +" = " + d + " * " +n.divide(d));
		}else
			System.out.println("Impossibile trovare una
fattorizzazione !");

	}//main

}
```

il cui output risulta essere il seguente:

```
Primo p1: 3883654937 lungo 32 bit
Primo p2: 3146459233 lungo 32 bit
Numero Composto n = p1*p2 : 12219761934309683321 lungo 64 bit
12219761934309683321 = 3883654937 * 3146459233
```

I metodi di fattorizzazione esposti, ovvero il *metodo di Fermat*, il *metodo ρ* e *p-1 di Pollard* sono stati testati in contemporanea nella classe *TestFactor.java*

```
package algebra;

import java.awt.*;
import java.awt.event.*;
import java.math.*;
import javax.swing.*;

public class TestFactor extends JFrame implements ActionListener {

        public static final BigInteger ONE=new BigInteger("1");
        Label promptLabel=new Label("Inserire un intero positivo da
fattorizzare:");
```

```
        TextArea bigNumArea=new TextArea("",
10,40,TextArea.SCROLLBARS_VERTICAL_ONLY);
        Panel buttonPanel=new Panel();
        Button FermatButton=new Button("Metodo di Fermat");
        Button PollardRhoButton=new Button("Metodo Rho di Pollard");
        Button pollardButton=new Button("Metodo p-1 di Pollard");
        TextArea factorArea=new TextArea("",
10,40,TextArea.SCROLLBARS_VERTICAL_ONLY);

        Thread th;

        public TestFactor() {

              Panel p1=new Panel();

              p1.setLayout(new GridLayout(4,1));
              p1.add(promptLabel);
              //numeri primi
              BigInteger primo = BigIntegerMath.getPrimeNbit(32);
              BigInteger secondo = BigIntegerMath.getPrimeNbit(32);
              // Crea un numero composto n = primo * secondo
              BigInteger n = primo.multiply(secondo);
              String val = n.toString();
              //bigNumArea.insert(val,0);
              bigNumArea.setText(val);
              p1.add(bigNumArea);
              buttonPanel.setLayout(new GridLayout(1,2));
              FermatButton.addActionListener(this);
              buttonPanel.add(FermatButton);
              PollardRhoButton.addActionListener(this);
              buttonPanel.add(PollardRhoButton);
              pollardButton.addActionListener(this);
              buttonPanel.add(pollardButton);
              p1.add(buttonPanel);
              p1.add(factorArea);
              factorArea.setEditable(false);

              add(p1);
              pack();
              show();

        }

        public void actionPerformed(ActionEvent e) {
           Object source=e.getSource();
           if (source==PollardRhoButton) {
              th=new Thread(
                 new Runnable() {
                    public void run() {
                       try {
                          BigInteger inputNum=new
BigInteger(bigNumArea.getText());
                          BigInteger
factor=BigIntegerMath.PollardRho(inputNum);
                          if (factor.compareTo(ONE) != 0 &&
factor.compareTo(inputNum) !=0)
                              factorArea.setText("Fattorizzazione Rho di
Pollard :\n"+inputNum +" = " + factor + " * " +inputNum.divide(factor));
                          else
                              factorArea.setText("Impossibile trovare una
fattorizzazione !");
```

```
                        } catch (NumberFormatException nfe) {
                           factorArea.setText(nfe.toString());
                        } catch (IllegalArgumentException iae) {
                           factorArea.setText(iae.toString());
                        }
                     }
                  }
               );
               th.start();
            } else if (source==pollardButton) {
               th=new Thread(
                  new Runnable() {
                     public void run() {
                        try {
                           BigInteger inputNum=new
BigInteger(bigNumArea.getText());
                           BigInteger
factor=BigIntegerMath.pMinusOneFactor(inputNum);
                           factorArea.setText("Fattorizzazione p-1 di Pollard :\
n"+inputNum +" = " + factor + " * " +inputNum.divide(factor));
                        } catch (NumberFormatException nfe) {
                           factorArea.setText(nfe.toString());
                        } catch (IllegalArgumentException iae) {
                           factorArea.setText(iae.toString());
                        }
                     }
                  }
               );
               th.start();
            } else if (source==FermatButton) {
              th=new Thread(
                        new Runnable() {
                              public void run() {
                                    try {
                                     BigInteger inputNum=new
BigInteger(bigNumArea.getText());
                                     //BigInteger
factor=BigIntegerMath.pMinusOneFactor(inputNum);
                                     BigInteger[] factors =
BigIntegerMath.fermat(inputNum);
                                     factorArea.setText("Fattorizzazione di
Fermat :\n"+inputNum +" = " + factors[0] + " * " +factors[1]);
                                    } catch (NumberFormatException nfe) {
                                       factorArea.setText(nfe.toString());
                                    } catch (IllegalArgumentException iae) {
                                       factorArea.setText(iae.toString());
                                    }
                              }
                              }
                        );
                        th.start();
                  }
         }

         public static void main(String[] arg){
                  new TestFactor();
            }//main

}
```

Generatori

Come noto un *gruppo* è una struttura algebrica formata da un insieme *G* con un'operazione binaria *, i cui elementi godono delle seguenti proprietà:

1. proprietà associativa : $\forall a,b,c \in G \mid (a*b)*c = a*(b*c)$
2. esistenza dell'elemento neutro : $\exists e \in G \mid \forall a \in G, a*e=e*a=a$
3. esistenza dell'elemento inverso : $\exists b \in G \mid \forall a \in G, a*b=b*a=e$

se *G* gode della proprietà commutativa questi si dirà *commutativo* o *abeliano*, cioè:

$$\forall a,b \in G \mid a*b=b*a$$

in particolare un gruppo si definisce *semplice* se non contiene al suo interno alcun *sottogruppo*[5] normale eccetto l'identità e se stesso. Sia $S \subseteq G$, indicheremo con $<S>$ il più piccolo sottogruppo di *G* che contiene *S*, questi viene chiamato *sottogruppo generato* da *S*.

La *cardinalità* dell'insieme *G* viene indicata con $|G|$ ed è chiamata *ordine* del gruppo: se questa è finita allora *G* è *finito*, altrimenti *infinito*. Risulta ovvio, che la tipologia di gruppo a cui noi siamo interessati nella nostra analisi è quella a cardinalità finita. Alcuni di questi tipi di gruppi sono caratterizzati dalla ciclicità, ovvero: un gruppo *G* è *ciclico* se esiste un elemento $g \in G$ detto *generatore*, tale che ogni altro elemento del gruppo lo si ottiene componendo *n* volte *g* con se stesso, scriveremo g^n oppure *ng*. In altre parole, l'insieme *G* coincide con il sottogruppo $<g>$ generato da *g*, il quale verrà indicato nella seguente maniera : $G = <g>$.

Un classico esempio di *gruppo ciclico* sono le *classi di resto* : si consideri il gruppo commutativo finito $Z/Z_n=\{0,1,2,...,n-1\}$ detto *gruppo quoziente*, in cui la somma fra due numeri *a* e *b* coincide con il resto delle divisione di $a+b$ per *n*. Inoltre, poiché ogni elemento lo si può scrivere come $n = 1+1+...+1$ per *n* volte, avremo che il numero *1* è un generatore del gruppo.

Se $g \in (G,*)$ definiamo *periodo di ordine G*, il più piccolo intero positivo *m* tale che $g^m = 1$ analogamente se $g \in (G,+)$ il periodo è definito dall'intero *m* tale che $mg = 1$. Da tale affermazione segue che il numero di generatori di un gruppo $G = <g>$ è pari a *m*.

A questo punto, ovviamente ci chiederemo : come si possono identificare i generatori o meglio ancora il generatore di un numero primo?

Supponiamo che sia *p* il nostro numero primo e $p_1^{e1}.p_2^{e2}....p_n^{en}$ la scomposizione in fattori primi, ottenuta magari con uno dei metodo prima esposti, del numero *p-1*, allora per identificare il generatore *g* modulo *p*, eseguiamo i seguenti passi:

1. Scegliamo un numero casuale intero *x* compreso tra 2 e p-2
2. $\forall i=1,...,n$
   - Calcoliamo $z = p/p_i$
   - Calcoliamo la congruenza $x^z \pmod p$
   - se il risultato è *1 modulo p*, allora *x* non è un generatore e ritorniamo al punto 1, altrimenti *x* è un generatore modulo *p*.

Come vedremo negli algoritmi a chiave pubblica, conoscere un generatore di un gruppo moltiplicativo di un campo finito risulterà assai utile, in quanto mediante l'uso di questi sarà possibile risalire alle chiavi utilizzate per la comunicazione. Pertanto è stata implementata una classe *PrimeGenerator.java* la quale ci permetterà di ottenere un numero primo ed il rispettivo

5 L'insieme *H* è un *sottogruppo* di *G* se e solo se è non vuoto ed è chiuso rispetto al prodotto ed all'inverso.

generatore, nella fattispecie tali valori verranno restituiti dal metodo *getSafePrimeAndGenerator()*, mentre solo il numero primo ci verrà comunicato dal metodo *getSafePrime()*.

```
package algebra;

import java.security.*;
import java.math.*;
import java.util.*;

/**
 * Permette di ottenere un numero primo.
 * */
public class PrimeGenerator {

      //Per trovare numeri primi, bisogna specificare la lunghezza minima di
quest'ultimi in bit
   int minBitLength;

   //Numero di test di primalistà
   int certainty;

   SecureRandom sr;

   /**
    * Costruttore
    * @param minBitLength di tipo int, rappresenta lunghezza minima in termini
di bit del numero primo
    * @param certainty di tipo int, rappresenta il numero di tentativi
    * @param sr di tipo SecureRandom, rappresenta l'oggetto che permette di
generare numeri casuali
    * @exception, nel caso in cui si cerchi di instanziare un numero primo
minore di 64 byte
    * */
   public PrimeGenerator(int minBitLength, int certainty, SecureRandom sr) {

         if (minBitLength<512)
        throw new IllegalArgumentException("Un numero primo sicuro deve essere
composto almeno da 64 byte.");
      //Associa i valori
      this.minBitLength=minBitLength;
      this.certainty=certainty;
      this.sr=sr;
   }

   /**
    * @return Restituisce un numero primo sicuro
    * */
   public BigInteger getStrongPrime() {
        //Il numero primo p sarà tale che p+1 sia composto da un grande fattore
s
      BigInteger s = new BigInteger(minBitLength/2-8,certainty,sr);
      //t will be a large prime factor of r, which follows
      //t sarà un grande fattore di r
      BigInteger t=new BigInteger(minBitLength/2-8,certainty,sr);
      BigInteger i=BigInteger.valueOf(1);
      //p-1 avrà un grande fattore r
      //r sarà il primo numero nella sequenza 2t+1, 2*2t+1, 2*3t+1,...
      BigInteger r;
      do {
         r=BigIntegerMath.TWO.multiply(i).multiply(t).add(BigIntegerMath.ONE);
         i=i.add(BigIntegerMath.ONE);
      } while (!r.isProbablePrime(certainty));
```

```
      BigInteger z=s.modPow(r.subtract(BigIntegerMath.TWO),r);
      BigInteger
pstar=BigIntegerMath.TWO.multiply(z).multiply(s).subtract(BigIntegerMath.ONE);
      BigInteger k=BigInteger.valueOf(1);
      //Il numero primo p è il primo nella sequenza 2rs+p*, 2*2rs+p*, 2*3rs
+p*,...
      BigInteger p=BigIntegerMath.TWO.multiply(r).multiply(s).add(pstar);
      while (p.bitLength()<=minBitLength) {
         k=k.multiply(BigIntegerMath.TWO);
         p=BigIntegerMath.TWO.multiply(k).multiply(r).multiply(s).add(pstar);
      }
      while (!p.isProbablePrime(certainty)) {
         k=k.add(BigIntegerMath.ONE);
         p=BigIntegerMath.TWO.multiply(k).multiply(r).multiply(s).add(pstar);
      }
      return p;
   }

   /**
    * @return un primo sicuro da 2rt+1 dove t è un grande primo
    * */
   public BigInteger getSafePrime() {
      BigInteger p;
      BigInteger r=BigInteger.valueOf(0x7fffffff);
      BigInteger t=new BigInteger(minBitLength-30,certainty,sr);
      //Il numero primo p è il primo nella sequenza 2rt+1, 2*2rt+1, 2*3rt+1,...
      do {
         r=r.add(BigIntegerMath.ONE);
         p=BigIntegerMath.TWO.multiply(r).multiply(t).add(BigIntegerMath.ONE);
      } while (!p.isProbablePrime(certainty));
      return p;
   }

   /**
    * Restituisce un primo sicuro della forma 2rt+1 dove t è un grande primo
    * e la fattorizzazione di r è nota.
    * Inoltre restituisce un generatore per il primo sicuro.
    * @return un array di due elementi di tipo BigInteger[], il quale nella
    * posizione 0 c'è il numero primo, mentre nella posizione 1 c'è il suo
    * generatore.
    * */
   public BigInteger[] getSafePrimeAndGenerator() {
      BigInteger[] p=new BigInteger[2];
      BigInteger r=BigInteger.valueOf(0x7fffffff);
      BigInteger t=new BigInteger(minBitLength-30,certainty,sr);
      //p è il numero primo nella sequenza 2rt+1, 2*2rt+1, 2*3rt+1,...
      do {
         r=r.add(BigIntegerMath.ONE);
         p[0]=BigIntegerMath.TWO.multiply(r).multiply(t).add(BigIntegerMath.ONE)
;
      } while (!p[0].isProbablePrime(certainty));

      //Bisogna ottenere i fattori primi da p-1=2rt
      //Inseriammo i fattori primi, presi una sola volta, in un Vector
      Vector factors=new Vector();

      //Aggiungiamo t nel vettore, ove t è un fattore primo di p-1=2rt
      factors.addElement(t);

      //Sappiamo che 2 è un fattore di p-1=2rt, così aggiungiamo 2 al vettore
      factors.addElement(BigInteger.valueOf(2));
```

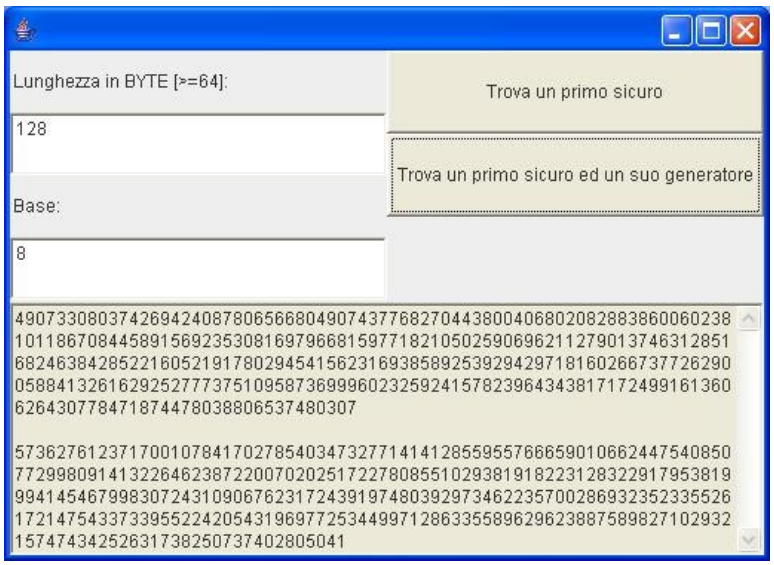
```
      //r potrebbe non essere un fattore
      if (r.isProbablePrime(10))
        factors.addElement(r);
      //diversamente, cerca i fattori di r e lo aggiungiamo al vettore
      else {
       //Riduce dutti e due i fattori di r e li aggiunge al vettore
         while (r.mod(BigIntegerMath.TWO).equals(BigIntegerMath.ZERO)) {
            r=r.divide(BigIntegerMath.TWO);
         }

         //Otteniamo ora i fattori primi di r, i quali dovrebbero essere
sufficientemente piccoli
         //partiamo con 3 - 2 che sono già nel vettore
         BigInteger divisor=BigInteger.valueOf(3);
         BigInteger square=divisor.multiply(divisor);
         while (square.compareTo(r)<=0) {
            //se questi divide r, lo aggiungiamo al vettore
            if (r.mod(divisor).equals(BigIntegerMath.ZERO)) {
               factors.addElement(divisor);
               //Riduce il divisore
               while (r.mod(divisor).equals(BigIntegerMath.ZERO))
                     r=r.divide(divisor);
            }
            divisor=divisor.add(BigIntegerMath.ONE);
            square=divisor.multiply(divisor);
         }

      }

      //Ora, partiamo con la ricerca del generatore
      boolean isGenerator;
      BigInteger pMinusOne=p[0].subtract(BigIntegerMath.ONE);
      BigInteger x,z,lnr;
      do {
       //Ipotiziamo che il vaore di test sia un generatore
         isGenerator=true;
         //Scegliamo un piccolo intero casuale x più piccolo del primo p
         x=new BigInteger(p[0].bitLength()-1,sr);
         for (int i=0;i<factors.size();i++) {
            //Calcoliamo z come p-1 dividendo con l'i-esimo fattore primo nel
vettore
            z=pMinusOne.divide((BigInteger)factors.elementAt(i));
            //Calcoliamo x^z(mod p)
            lnr=x.modPow(z,p[0]);
            //Se questi è pari ad 1, non è un generatore
            if (lnr.equals(BigIntegerMath.ONE)) {
               isGenerator=false;
               //Cerchiamo un nuovo generatore
               break;
            }
         }
         //Finche x non è un generatore, ritorniamo indietro e generiamo un
nuovo x
      } while (!isGenerator);
      //Abbiamo trovato un insieme di generatori, e li restituiamo
      p[1]=x;
      return p;
   }

}
```

Tale metodologia è stata testata con la classe *TestPrimeGenerator.java*.

```
package algebra;

import java.awt.*;
import java.awt.event.*;
import java.math.*;
import java.security.*;
import javax.swing.*;

public class TestPrimeGenerator extends JFrame implements ActionListener {

  Label byteLengthLabel=new Label("Lunghezza in BYTE [>=64]:");
  TextField byteLengthField=new TextField();
  Label numberBasesLabel=new Label("Base:");
  TextField numberBasesField=new TextField();
  Button getStrongPrimeButton=new Button("Trova un primo sicuro");
  Button getSafePrimeButton=new Button("Trova un primo sicuro ed un suo
generatore");
  TextArea primeArea=new TextArea("",10,40,TextArea.SCROLLBARS_VERTICAL_ONLY);
  Panel p1=new Panel();
  Panel p2=new Panel();
  Panel p3=new Panel();
  Panel p4=new Panel();

  SecureRandom sr;
  int byteLength=0;
  int numberBases=0;

  public TestPrimeGenerator() {

        Panel p=new Panel();

        p.setLayout(new GridLayout(2,1));
        p.add(p1);
        p1.setLayout(new GridLayout(1,2));
        p1.add(p3);
        p3.setLayout(new GridLayout(4,1));
        p1.add(p4);
        p4.setLayout(new GridLayout(3,1));
        p.add(p2);
        p2.setLayout(new GridLayout(1,1));

        p3.add(byteLengthLabel);
        p3.add(byteLengthField);
        p3.add(numberBasesLabel);
        p3.add(numberBasesField);
        p4.add(getStrongPrimeButton);
        getStrongPrimeButton.addActionListener(this);
        p4.add(getSafePrimeButton);
        getSafePrimeButton.addActionListener(this);
        p2.add(primeArea);
        primeArea.setEditable(false);
        sr=new SecureRandom();

        add(p);
        pack();
        show();
  }

  public void actionPerformed(ActionEvent e) {
    Object culprit=e.getSource();
```

```
    if (culprit==getStrongPrimeButton) {
      primeArea.setText("Attendere prego ...");
      try {
        byteLength=Integer.parseInt(byteLengthField.getText());
        if (byteLength<=0) {
          byteLength=1;
          byteLengthField.setText("1");
          primeArea.setText("La lunghezza in BYTE deve essere un numero intero
positivo!");
        }
        numberBases=Integer.parseInt(numberBasesField.getText());
        PrimeGenerator pg=new PrimeGenerator(8*byteLength,numberBases,sr);
        BigInteger num=pg.getStrongPrime();
        primeArea.setText(num.toString());
      } catch (NumberFormatException nfe) {
        primeArea.setText(nfe.toString());
      } catch (IllegalArgumentException iae) {
        primeArea.setText(iae.toString());
      }
    } else if (culprit==getSafePrimeButton) {
      primeArea.setText("Attendere prego ...");
      try {
        byteLength=Integer.parseInt(byteLengthField.getText());
        if (byteLength<=0) {
          byteLength=1;
          byteLengthField.setText("1");
          primeArea.setText("La lunghezza in BYTE deve essere un numero intero
positivo!");
        }
        numberBases=Integer.parseInt(numberBasesField.getText());
        PrimeGenerator pg=new PrimeGenerator(8*byteLength,numberBases,sr);
        BigInteger[] nums=pg.getSafePrimeAndGenerator();
        primeArea.setText(nums[0].toString()+"\n\n"+nums[1].toString());
      } catch (NumberFormatException nfe) {
        primeArea.setText(nfe.toString());
      } catch (IllegalArgumentException iae) {
        primeArea.setText(iae.toString());
      }
    }
  }

  public static void main(String[] arg){
          new TestPrimeGenerator();
    }//main

}
```

il cui output risulta essere il seguente:



Calcolo del Logaritmo Discreto

Consideriamo un gruppo finito G di ordine n ed un elemento $b \in G$. Posto

$$S = Z_n = \{0,1, \dots , n-1\}$$

possiamo considerare la *funzione esponenziale*

$$f: S \rightarrow G \, , \, f(x) = b^x$$

Se $y=f(x)$, si dice che x è un *logaritmo discreto* su G in base b e si indica con il simbolo $log_b\, y$. Nel caso in cui G sia il gruppo moltiplicativo F^*_q di un campo finito F_q , detto $b \in F^*_q$ un suo generatore, allora f è biettiva e la sua funzione inversa si dice *logaritmo discreto* su F_q in base b. Si noti, che in questo caso tale operazione la si effettua in tempo *polinomiale*, mentre l'inversa, ovvero il costo del calcolo del logaritmo discreto è di tipo *esponenziale* e si congettura che non esistano algoritmi che riescano a risolvere tale problema in tempo polinomiale. Ecco perché tale asimmetria computazionale risulta essere lo strumento idoneo per la realizzazione di un crittosistema a chiave asimmetrica!

Tuttavia si evidenzia che esistono alcuni algoritmi, alquanto efficaci, per il calcolo di logaritmi discreti come quello descritto nel paragrafo a seguire.

Algoritmo Baby step – Giant step

Il dominio di azione di tale algoritmo è il gruppo moltiplicativo Z^*_p i cui elementi li identifichiamo con $1,2, \dots ,p-1$. Sia g un generatore in Z^*_p. Vogliamo determinare il logaritmo discreto x di $y \in Z_p$ in base g. Si procede nel modo seguente:

- *baby steps* : si pone n uguale al minimo intero maggiore di $\sqrt{p}$ e si calcolano i valori $g^i \in \{1, \dots ,p-1\}$ per ogni $i \in \{0, \dots ,n-1\}$, ponendoli poi in una lista da tenere in memoria;
- *giant steps* : si calcolano g^n e poi g^{-n} e successivamente yg^{-n}, yg^{-2n} , yg^{-3n} , ... , $yg^{-n\wedge 2}$. Dopo ognuno di questi calcoli confrontiamo i risultati ottenuti con i numeri della lista ottenuta nel primo passo. Appena ottenuta un'uguaglianza del tipo $yg^{-jn} = g^i$, abbiamo trovato il logaritmo $x = jn+i$.

Notiamo innanzitutto che l'algoritmo termina senz'altro producendo il logaritmo richiesto. Infatti, tale logaritmo esiste ed è un numero x in $\{1, \dots , p-1\}$. Quindi dividendo x per n si ha $x = nj+i$, con $0 \leq i \leq n-1$. D'altra parte poiché $x < p < n^2$ si ha pure $1 \leq j \leq n$.

In secondo luogo, stimiamo la complessità dell'algoritmo. I passi compiuti durante i *baby steps* sono *n* ed ognuno ha complessità *O(log n)*. Quindi la complessità totale è *O(n log n)*. Similmente, si vede anche che i passi compiuti durante i *giant steps*, i quali dipendono dal numero di cui si cerca il logaritmo, hanno complessità *O(n log n)* e questa dunque la complessità dell'algoritmo. Essa è esponenziale in *n*.

Le problematiche riscontrate dall'utilizzo di tale metodologia sono le seguenti:

1. l'algoritmo richiede un enorme consumo di memoria se p è molto grande, poiché occorre memorizzare una lista di $[\sqrt{p}]+1$ interi;
2. il confronto tra gli elementi delle due liste baby e giant steps ha un suo costo computazionale, il quale in prima istanza di analisi è stato trascurato. Un modo per effettuare il confronto consiste nel dividere i due numeri ed analizzarne il quoziente, ovvero accertarsi che questi sia o meno maggiore di zero. Risulta chiaro che l'algoritmo che siamo portati a svolgere compie n^2 confronti tra numeri e dunque anche l'insieme di tali confronti ha complessità esponenziale.

Ad esempio si ipotizzi di voler attuare il procedimento illustrato con $p = 11$ e $g = 2$ che è un generatore di Z^*_{11}. In questo caso abbiamo che $3 < \sqrt{11} < 4$ e quindi $n = 4$. I baby steps portano alla creazione della lista $2^x=r \mid x = \{0,1,2,3\}$:

- $2^0 = 1$
- $2^1 = 2$
- $2^2 = 4$
- $2^3 = 8$

si calcola poi $2^4 = 16 \equiv 5 (mod\ 11)$, mentre il suo inverso vale $9 (mod\ 11)$.

Supponiamo di voler calcolare il logaritmo $log_2\ 6$. Si procede allora con i giant steps. Il primo passo ci porta a calcolare a $6 \times 2^{-4} = 54 \equiv 10 (mod\ 11)$. Poiché questo numero non è contenuto nella lista dei baby step, non abbiamo trovato il logaritmo richiesto. Effettuiamo un altro giant step, calcoliamo $6 \times 2^{-8} \equiv 2 (mod\ 11)$. Scopriamo allora la relazione $6 \times 2^{-8} = 2$ ossia $6 = 2^9 (mod\ 11)$, cioè $log_2\ 6 = 9$.

I metodi precedentemente esposti sono stati implementati nella classe *BigIntegerMath.java*.

```
 /**
 * Risolve la congruenza g^x = r(modulo p) ove g è il generatore di p
 * mediante una ricerca esaustiva, ovvero sostituendo in x i valori
 * compresi tra 0 e p-1
 * @param base di tipo BigInteger, rappresenta il generatore
 * @param residue di tipo BigInteger, rappresenta il resto residuo
 * @param modulus di tipo BigIntegr, rappresenta il modulo
 * @return un valore di tipo BigInteger
 * */
public static BigInteger logExhaustiveSearch(BigInteger base, BigInteger
residue, BigInteger modulus) {

	BigInteger basePow=BigInteger.valueOf(1);
	BigInteger j;
	for (j=BigInteger.valueOf(1);j.compareTo(modulus)<0;j=j.add(ONE)) {
		basePow=basePow.multiply(base).mod(modulus);
		if (basePow.equals(residue))
			break;
	}//for
	if (j.equals(modulus))
		throw new NoSuchElementException("Nessuna soluzione!");
```

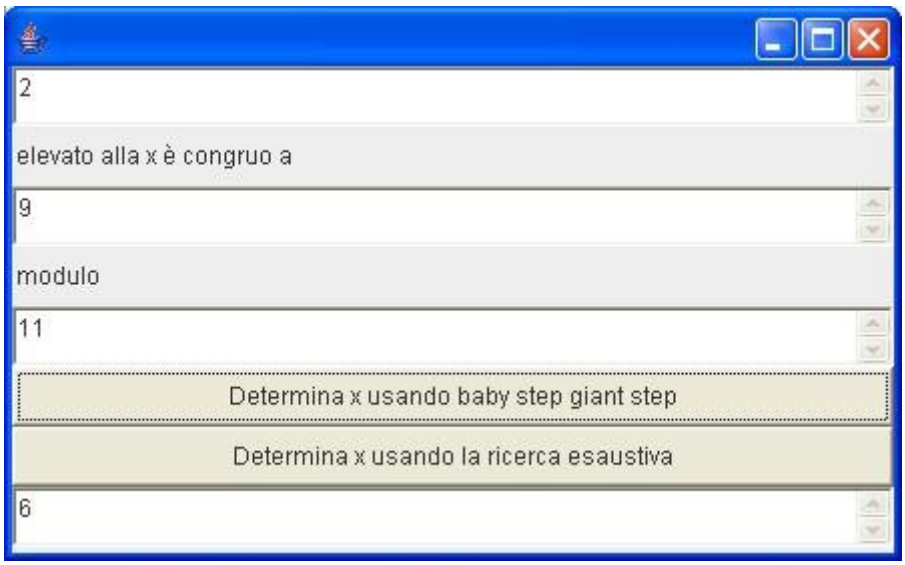
```
      return j;

}//logExhaustiveSearch

/**
 * Risolve la congruenza g^x = r(modulo p) ove g è il generatore di p
 * mediante l'algoritmo baby steps - giant steps
 * @param base di tipo BigInteger, rappresenta il generatore
 * @param residue di tipo BigInteger, rappresenta il resto residuo
 * @param modulus di tipo BigIntegr, rappresenta il modulo
 * @return un valore di tipo BigInteger
 * */
public static BigInteger logBabyStepGiantStep(BigInteger base, BigInteger
residue, BigInteger modulus) {

      BigInteger m=sqrt(modulus).add(ONE);
      //Usa una hsh table per memorizzare i valori in uso
      Hashtable h=new Hashtable();
      BigInteger basePow=BigInteger.valueOf(1);
      //Crea l'hash table con chiave base^j, l'indice j è il valore
      for (BigInteger j=BigInteger.valueOf(0);j.compareTo(m)<0;j=j.add(ONE)) {
            h.put(basePow,j);
            basePow=basePow.multiply(base).mod(modulus);
      }
      //Calcola l'inverso di base^m modulo p
      BigInteger basetotheminv=base.modPow(m,modulus).modInverse(modulus);
      BigInteger y=new BigInteger(residue.toByteArray());
      //Effettua una ricerca nell'hash table per un base^j tale che y=base^j per
qualche j
      BigInteger target;
      for (BigInteger i=BigInteger.valueOf(0);i.compareTo(m)<0;i=i.add(ONE)) {
            target = (BigInteger)h.get(y);
            if (target!=null)
                  return i.multiply(m).add(target);
            y=y.multiply(basetotheminv).mod(modulus);
      }//for
      throw new NoSuchElementException("Nessuna soluzione");
}//logBabyStepGiantStep
```

e testati con la lasse *TestDiscreteLog.java*

```
package algebra;

import java.awt.*;
import java.awt.event.*;
import java.math.*;
import javax.swing.*;

public class TestDiscreteLog extends JFrame implements ActionListener {

   TextArea baseArea=new TextArea();
   Label l1=new Label("elevato alla x è congruo a");
   TextArea resArea=new TextArea();
   Label l2=new Label("modulo");
   TextArea modArea=new TextArea();
   Button bsgsButton=new Button("Determina x usando baby step giant step");
   Button esButton=new Button("Determina x usando la ricerca esaustiva");
   TextArea logArea=new TextArea();

   BigInteger base;
   BigInteger residue;
   BigInteger modulus;
```

```
   public TestDiscreteLog() {

         Panel p1=new Panel();

         p1.setLayout(new GridLayout(8,1));
         p1.add(baseArea);
         p1.add(l1);
         p1.add(resArea);
         p1.add(l2);
         p1.add(modArea);
         p1.add(bsgsButton);
         bsgsButton.addActionListener(this);
         p1.add(esButton);
         esButton.addActionListener(this);
         p1.add(logArea);

         add(p1);
         pack();
         show();
   }

   public void actionPerformed(ActionEvent e) {
      try {
         base=new BigInteger(baseArea.getText());
         residue=new BigInteger(resArea.getText());
         modulus=new BigInteger(modArea.getText());
         if (e.getSource()==bsgsButton) {
            logArea.setText(BigIntegerMath.logBabyStepGiantStep(base,residue,mod
ulus).toString());
         } else if (e.getSource()==esButton) {
            logArea.setText(BigIntegerMath.logExhaustiveSearch(base,residue,modu
lus).toString());
         }
      } catch (Exception exc) {
         logArea.setText(exc.toString());
      }
   }

   public static void main(String[] arg){
            new TestDiscreteLog();
      }//main

}
```

il cui output risulta essere il seguente:


2
elevato alla x è congruo a
9
modulo
11
Determina x usando baby step giant step
Determina x usando la ricerca esaustiva
6

CAPITOLO 2
Il modello Informatico

Introduzione

In questo capitolo vengono proposti alcuni dei principali modelli crittografici utilizzati e consolidati nel mondo informatico. L'illustrazione di tali strutture verrà corredata sia del rispettivo modello matematico che implementativo in codice Java. Tali strutture di carattere algebrico, si serviranno nell'implementazione degli algoritmi crittografici proposti nel *capitolo 1*.

Cos'è la crittografia

Definizione : la crittografia (dal greco *kryptos*, nascosto, e *graphein*, scrivere) è la scienza che si occupa dello studio delle scritture segrete, utilizzando tecniche di teoria dei numeri e di teoria dell'informazione. Questi si differenzia dalla *Steganografia*, la quale è una scienza simile ma sostanzialmente diversa, la quale si occupa dello studio di come nascondere un informazione all'interno di un altro dato.

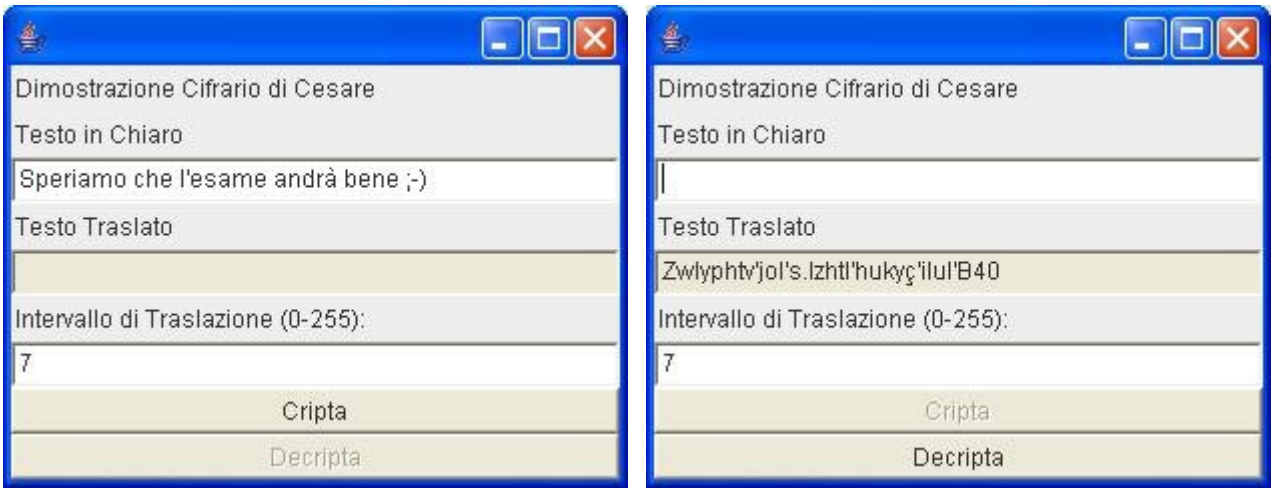
La crittografia è un metodo per trasformare le informazioni in una forma tale che, anche se una persona diversa dal destinatario riuscisse ad avere accesso ai dati, non potrebbe comprenderne il significato. Oggi la crittografa è utilizzata per lo scambio di messaggi tra due soggetti ma anche per altri scopi: ad esempio per cifrare un file memorizzato all'interno di un disco di un computer.

Cifratura e crittografa sono termini comunemente usati che hanno il medesimo significato; non a caso il famoso metodo di Giulio Cesare, precursore delle moderne tecniche di crittografia, è chiamato appunto *cifrario* mediante il quale trasmetteva i suoi messaggi con tre cifre spostate.

```
package cifrari;

import java.math.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Implementa il Cifrario di Cesare.
 * Come noto, questi risulta essere uno dei sistemi di criptaggio
 * più semplici da realizzare, in quanto basato su traslazione delle
 * lettere. L'afabeto considerato è quello ASCII da 0-255, per ogni
 * carattere viene effettuata una traslazione su ogni singolo bit.
 * */
public class CaesarCipher extends JFrame implements ActionListener{

      int shift=0;
      byte[] msgArray=null;
      byte[] encmsgArray=null;
      Label titleLabel=new Label("Dimostrazione Cifrario di Cesare");
      Label Label1=new Label("Testo in Chiaro");
```

```
	TextField msg=new TextField(40);
	Label Label2=new Label("Testo Traslato");
	TextField encmsg=new TextField(40);
	Label shiftLabel=new Label("Intervallo di Traslazione (0-255):");
	TextField entryShiftValue=new TextField(40);
	Button encipherButton=new Button("Cripta");
	Button decipherButton=new Button("Decripta");

	//costruttore
	public CaesarCipher() {

		Panel p1=new Panel();

		p1.setLayout(new GridLayout(9,1));
		p1.add(titleLabel);
		p1.add(Label1);
		p1.add(msg);
		p1.add(Label2);
		p1.add(encmsg);
		encmsg.setEditable(false);
		p1.add(shiftLabel);
		p1.add(entryShiftValue);
		p1.add(encipherButton);
		encipherButton.addActionListener(this);
		p1.add(decipherButton);
		decipherButton.addActionListener(this);
		decipherButton.setEnabled(false);
		add(p1);
		pack();
		show();
	}//CaesarCipher

	public void actionPerformed(ActionEvent e) {

		if (e.getSource()==encipherButton) {
			try {
				shift=Integer.parseInt(entryShiftValue.getText());
			} catch (NumberFormatException nfe) {
				shift=0;
			}//catch

			msgArray=msg.getText().getBytes();
			encmsgArray=caesarEncipher(msgArray,shift);
			encmsg.setText(new String(encmsgArray));
			msg.setText("");
			encipherButton.setEnabled(false);
			decipherButton.setEnabled(true);
		} else if (e.getSource()==decipherButton) {
			msgArray=caesarDecipher(encmsgArray,shift);
			msg.setText(new String(msgArray));
			encmsg.setText("");
			decipherButton.setEnabled(false);
			encipherButton.setEnabled(true);
		}//else if

	}//actionPerformed

	/**
	 * Cripta il messaggio traslando il suo contenuto di <i>shift</i>
caratteri.
	 * @param message di tipo byte[], rappresenta il messaggio in chiaro
```

```
	 * @param shift di tipo int, rappresenta chiave di traslazione
	 * @return un array di byte rappresentante il testo cifrato.
	 */
	private static byte[] caesarEncipher(byte[] message,int shift) {

		byte[] m2=new byte[message.length];
		for (int i=0;i<message.length;i++) {
			m2[i]=(byte)((message[i]+shift)%256);
		}//for

		return m2;
	}//caesarEncipher

	/**
	 * Il metodo di decriptaggio.
	 * @param message di tipo byte[], rappresenta il messaggio criptato.
	 * @param shift di tipo int, rappresenta la chiave di traslazione.
	 * @return un array di byte rappresentante il testo decifrato.
	 */
	private static byte[] caesarDecipher(byte[] message,int shift) {

		byte[] m2=new byte[message.length];
		for (int i=0;i<message.length;i++) {
			m2[i]=(byte)((message[i]+(256-shift))%256);
		}//for

		return m2;
	}//caesarDecipher

	public static void main(String[] arg){
		new CaesarCipher();
	}//main

}//CaesarCipher
```



Uno dei principi fondamentali del moderni sistemi di crittografia è stato individuato nel 1883 dal francese August Kerckhoffs, il quale affermava che la segretezza di un messaggio cifrato deve risiedere interamente nella *chiave* utilizzata e non nel metodo di cifratura, che può essere noto! Questo avviene oramai per buona parte degli algoritmi crittografici attuali.

Eiste una vera e propria scienza dedicata alla scardinatura di sistemi di crittografia, detta *crittoanalisi*. La tecnica di crittonalisi più nota risulta quella basata sulla c.d. *forza bruta*, ossia sul

calcolo di tutte le possibili combinazioni di chiavi del cifrario. Con l'aumento della potenza di calcolo degli elaboratori, questa tecnica banale sta diventando sempre più efficace. Basti pensare che l'algoritmo DES con chiave a 56bit, per esempio, è stato violato nel 1999 in sole 22 ore e 15 minuti con uno speciale computer multi-processore. Questo non significa che il DES sia poco affidabile: è la sua chiave che è relativamente corta! Più lunga è la chiave, maggiore è la sicurezza offerta dal sistema di crittografia. Non a caso gli altri algoritmi considerati più sicuri del DES hanno chiavi molto più lunghe. Lo stesso Triple DES usa una chiave più lunga. La *crittologia* è la scienza che studia la crittografia e la crittoanalisi.

La crittografia è un vero e proprio processo, che inizia con un testo in chiaro, detto *Plain Text*, e termina con lo stesso testo in chiaro, bypassando per uno stato di cifratura detto *Chiper Text*.

Come possiamo notare dall'immagine soprastante, le operazioni di trasformazione sono denominate:

- *cifratura* : quando un messaggio originale viene codificato;
- *decifratura* : quando il messaggio codificato viene riportato nella sua versione originale.

Nello scambio di messaggi, i rischi di sicurezza classici sono le frodi dovute all'impersonificazione del mittente, l'intercettazione della comunicazione, le modifiche del contenuto dei messaggi, la possibilità di negare la trasmissione o ricezione di un messaggio.

Per contrastare questi rischi è necessario predisporre misure che garantiscono i noti requisiti di *integrità* e *riservatezza*, con l'aggiunta di altri due, l'*autenticità* ed il *non ripudio* che consiste nella prevenzione della possibilità di negare la trasmissione o la ricezione delle informazioni.

La crittografia è oggi una soluzione affermata per soddisfare questi requisiti. Esistono sostanzialmente due tipologie di sistemi di crittografia moderna: uno a *chiave segreta* o *simmetrica* e l'altro a *chiave pubblica* o *asimettrica*.

Impostazione matematica di un crittosistema

Per descrivere matematicamente un crittosistema, il modo più efficace è quello di associare ad ogni simbolo del nostro alfabeto un *numero intero*. Per fissare le idee, supponiamo che l'alfabeto in cui scriviamo i nostri messaggi sia l'alfabeto della lingua inglese e ne consideriamo gli equivalenti numerici.

$a \to 0$	$h \to 7$	$o \to 14$	$u \to 20$
$b \to 1$	$i \to 8$	$p \to 15$	$v \to 21$
$c \to 2$	$j \to 9$	$q \to 16$	$w \to 22$
$d \to 3$	$k \to 10$	$r \to 17$	$x \to 23$
$e \to 4$	$l \to 11$	$s \to 18$	$y \to 24$
$f \to 5$	$m \to 12$	$t \to 19$	$z \to 25$
$g \to 6$	$n \to 13$		

Poiché usiamo un alfabeto con 26 lettere risulta naturale effettuare tutte le operazioni matematiche sugli equivalenti numerici delle lettere modulo 26. in generale se il messaggio unitario è *un blocco di s lettere* $a_1, a_2 ..., a_s$ allora possiamo etichettare il messaggio unitario $a_1, a_2 ..., a_s$ con la struttura

di interi $x_1, x_2 ..., x_s$ dove x_i con $1 \leq i \leq s$ è l'equivalente numerico di a_i.

Con la tecnica appena illustrata risulta essere caratterizzata da un problema di ambiguità di scrittura su cui è bene porre l'accento.

Supponiamo infatti di voler trasmettere un messaggio con blocchi di due elementi e di usare l'equivalente numerico delle lettere. Allora la sequenza numerica *114* potrebbe corrispondere al messaggio *bo* ma anche al messaggio *le* a seconda che il numero *114* lo si guardi come composto da *1* e *14* oppure *11* e *4*. Tutto questo perché la corrispondenza numerica viene effettuata con numeri che non hanno sempre lo stesso numero di cifre.

Il problema appena esposto può essere risolto utilizzando la corrispondenza binaria o quella numerica a due cifre.

	Eq. num.	A 2 cifre	Binario
le	114	1104	0101100100
bo	114	0114	0000101110

Per quanto riguarda gli algoritmi proposti in seguito, verrà utilizzato il codice *ASCII (American Standard Code for Information Interchange)* come modo standard di associare ad ogni lettera dell'alfabeto l'equivalente numerico.

```
33 !  34 "  35 #  36 $  37 %  38 &  39 '  40 (  41 )  42 *
43 +  44 ,  45 -  46 .  47 /  48 0  49 1  50 2  51 3  52 4
53 5  54 6  55 7  56 8  57 9  58 :  59 ;  60 <  61 =  62 >
63 ?  64 @  65 A  66 B  67 C  68 D  69 E  70 F  71 G  72 H
73 I  74 J  75 K  76 L  77 M  78 N  79 O  80 P  81 Q  82 R
83 S  84 T  85 U  86 V  87 W  88 X  89 Y  90 Z  91 [  92 \
93 ]  94 ^  95 _  96 `  97 a  98 b  99 c  100 d 101 e 102 f
103 g 104 h 105 i 106 j 107 k 108 l 109 m 110 n 111 o 112 p
113 q 114 r 115 s 116 t 117 u 118 v 119 w 120 x 121 y 122 z
123 { 124 | 125 } 126 ~ 127 □ 128 ? 129 ? 130 ? 131 ? 132 ?
133 ? 134 ? 135 ? 136 ? 137 ? 138 ? 139 ? 140 ? 141 ? 142 ?
143 ? 144 ? 145 ? 146 ? 147 ? 148 ? 149 ? 150 ? 151 ? 152 ?
153 ? 154 ? 155 ? 156 ? 157 ? 158 ? 159 ? 160    161 ¡ 162 ¢
163 £ 164 ¤ 165 ¥ 166 ¦ 167 § 168 ¨ 169 © 170 ª 171 « 172 ¬
173   174 ® 175 ¯ 176 ° 177 ± 178 ² 179 ³ 180 ´ 181 µ 182 ¶
183 · 184 ¸ 185 ¹ 186 º 187 » 188 ¼ 189 ½ 190 ¾ 191 ¿ 192 À
193 Á 194 Â 195 Ã 196 Ä 197 Å 198 Æ 199 Ç 200 È 201 É 202 Ê
203 Ë 204 Ì 205 Í 206 Î 207 Ï 208 Ð 209 Ñ 210 Ò 211 Ó 212 Ô
213 Õ 214 Ö 215 × 216 Ø 217 Ù 218 Ú 219 Û 220 Ü 221 Ý 222 Þ
223 ß 224 à 225 á 226 â 227 ã 228 ä 229 å 230 æ 231 ç 232 è
233 é 234 ê 235 ë 236 ì 237 í 238 î 239 ï 240 ð 241 ñ 242 ò
243 ó 244 ô 245 õ 246 ö 247 ÷ 248 ø 249 ù 250 ú 251 û 252 ü
253 ý 254 þ 255 ÿ
```

Tale tabulato può essere facilmente realizzato con l'esecuzione della classe *DisplayCharacterSet.java* del package *cifrari*.

```
package cifrari;

/**
 * Restituisce i codici ASCII dei caratteri
 */
public class DisplayCharacterSet {
      public static void main(String[] args) {
            int count = 1;
            for (int i=33;i<256;i++){
                  if(count%10==0){
                        System.out.print(i+" "+(char)i+"\t");
```

```
                        System.out.println();
                    }else{
                        System.out.print(i+" "+(char)i+"\t");
                    }//else
                    count++;
                }//for
        }
}
```

La crittografa a chiave segreta o simmetrica

Una persona *(mittente)* che vuole inviare un messaggio cifrato ad un'altra *(destinatario)*, può usare un sistema di cifratura basato su una chiave segreta. La stessa chiave è poi usata dal destinatario del messaggio per decifrarlo e riportarlo quindi in chiaro.

Quest'operazione presuppone che entrambi, mittente e destinatario, conoscano la chiave segreta. La crittografia a chiave segreta o crittografia simmetrica, richiede che il mittente ed il destinatario siano d'accordo con la chiave da condividere ed usino un metodo sicuro per scambiarsela.

Nasce così un problema: coma fa il mittente a far pervenire in modo sicuro, la chiave segreta al destinatario? Se la invia tramite posta elettronica ad esempio, qualcuno potrebbe intercettarla; se la comunicazione per telefono non ha la garanzia dell'identificazione del destinatario. Si potrebbe consegnare a mano, ma i costi dell'operazione potrebbero essere eccessivi. Inoltre si consideri che esiste anche un limite sulla quantità di chiavi gestibili, poiché per ogni possibile coppia di utenti deve esistere una chiave segreta: per n utenti occorrono $[n(n-1)/2]$ chiavi. Se gli utenti che devono scambiarsi un messaggio sono 10, le chiavi da scambiarsi in modo sicuro sono 45. Quindi in linea mi massima possiamo asserire che *le chiavi crescono col quadrato del numero degli utenti*!

La crittografia a chiave asimmetrica al contrario non richiede una distribuzione sicura delle chiavi, in quanto quella da scambiare è la pubblica (lo dice il nome!), non quella privata, che resta sotto la responsabilità del solo utente proprietario.

Anticipando alcune caratteristiche, sembrerebbe che la crittografia asimmetrica sia da preferire rispetto alla simmetrica; in realtà presentano entrambi vantaggi e svantaggi. Il sistema a chiave segreta ha il vantaggio di essere più indicato per la cifratura di grossi volumi di dati ma inadatto per lo scambio sicuro di messaggi in una popolazione di utenti numerosi. Il metodo a chiave pubblica ha un processo di elaborazione più lungo, inadatto per la cifratura di grossi volumi di dati, ma vantaggioso per molte altre operazioni, tipo l'identificazione delle parti coinvolte in una comunicazione. Tale aspetto verrà chiarito meglio nei paragrafi a seguire.

Per quanto riguarda la garanzia dei requisiti nello scambio di informazioni, la crittografia a chiave segreta assicura la riservatezza, mentre l'integrità è garantita in maniera indiretta: se non si riesce a decifrare il messaggio con la chiave segreta, allora il messaggio è stato alterato. In ogni caso il requisito di integrità non può essere garantito in modo separato da quello di riservatezza.

Anche l'autenticità ed il non ripudio sono garantiti in modo indiretto, poiché si assume che solo un'altra persona possa conoscere la chiave segreta, quindi deve essere per forza questa ad averci inviato il messaggio. Questo vale unicamente in situazioni di scambio di informazioni in una popolazione di sole due persone, in assenza di compromissione della chiave.

Cifrari basati su aritmetica modulare

Ipotizziamo di dover inviare il nostro messaggio, mediante l'uso di una rete a pacchetti (blocchi) come ad esempio la rete Internet, proseguiremo nella seguente maniera:

- fissiamo la lunghezza del messaggio unitario, aggiungendo se necessario alla fine del messaggio la lettera *x* un numero sufficientemente di volte per far si che il messaggio possa essere diviso in blocchi della stessa lunghezza;
- trasformiamo i vari blocchi in equivalenti numerici;
- scegliamo una chiave *k* ed un cifrario corrispondente: definiamo cioè la funzione C_k che determina il cifrario sull'alfabeto dei numeri;
- determiniamo $D_k=C_k^{-1}$;
- ricostruiamo il messaggio nell'alfabeto solito.

Risulta chiaro quindi che un'operazione di cifratura altro non è che una funzione invertibile a valori in $\mathbb{Z}_N^s$ ove N è il numero di caratteri dell'alfabeto, per esempio $N=26$ se consideriamo le lettere dell'alfabeto inglese, mentre s è il numero di caratteri che compongono il blocco.

$$f : \mathbb{Z}_N^s \rightarrow \mathbb{Z}_N^s$$

Pertanto, il blocco di due lettere *le* ed *bo*, sul quale prima era stato sollevato il problema dell'ambiguità potrà essere cifrato nella seguente maniera:

$$le \rightarrow 26 \cdot 11 + 4 = 290 \qquad bo \rightarrow 26 \cdot 1 + 14 = 40$$

In definitiva, tale tipo di cifrario a chiave segreta, è uno dei cosiddetti *cifrari classici*, il quale è caratterizzato da una chiave D_k per decifrare, la quale si calcola facilmente a partire dalla chiave C_k usata per cifrare. In altri termini, dal punto di vista computazionale, la conoscenza della chiave per decifrare *è sostanzialmente equivalente* a quella della chiave per cifrare. Nei cifrari a chiave pubblica, o cosiddetti *a due vie* i quali verranno descritti più in avanti, sono basati su idee matematiche molto diverse, si può al contrario rendere nota la chiave per cifrare senza che questo comprometta la segretezza di D_k. Infatti in questi sistemi il calcolo di D_k a partire da C_k è tanto arduo dal punto di vista computazionale, da renderlo in pratica impossibile.

Negli esempi che seguiranno, per semplicità rappresentativa supporremo che il messaggio unitario sia costituito da una sola lettera, cioè ipotizzeremo che l'alfabeto numerico dei messaggi sia $P = Z_{26}$. Se P è una qualunque lettera dell'alfabeto, continueremo ad indicare con P anche il suo equivalente numerico. Useremo la stessa convenzione anche per la lettera C del testo cifrato.

Cifrari affini

I cifrari affini sono descritti da una funzione cifrante che utilizza una *trasformazione affine*, ossia una funzione biettiva:

$$C_k : Z_N \rightarrow Z_N \qquad P \rightarrow (aP + b) \bmod N$$

Dove $a,b \in Z$ e la coppia $k = (a,b)$ rappresenta la *chiave* del sistema. Perché C_k sia biettiva, è necessario che a sia relativamente primo con N (nell'esempio pratico considereremo N=26 quale cardinalità dell'alfabeto inglese), cioè che $MCD(a,N)=1$. In tal caso la congruenza $xa \equiv 1 \pmod N$ ha un'unica soluzione $a' \pmod N$. Allora la funzione inversa di decifratura D_k sarà

$$D_k = C_k^{-1} : Z_N \rightarrow Z_N \qquad C \rightarrow a'(C - b) \bmod N$$

Osserviamo che i *cifrari di Cesare* o di *traslazione*, presentati in precedenza, sono cifrari affini con

$a=1$.

Supponendo di lavorare in Z_{26} possiamo supporre $0 \leq a,b \leq 25$, prendendo a e b modulo 26. Ricordando che a è relativamente primo con 26 se, e solo se, a è un elemento invertibile in Z_{26} , ossia a $\in U(Z_{26})$, ne segue che l'insieme delle chiavi è

$$K = U(Z_{26}) \times Z_{26}$$

Ma questo punto risulta lecito chiedersi:

1. quanti sono i cifrari affini?
2. Quanti elementi ha K?

Per rispondere a queste due domande ci viene incontro la *funzione di Eulero* così definita:

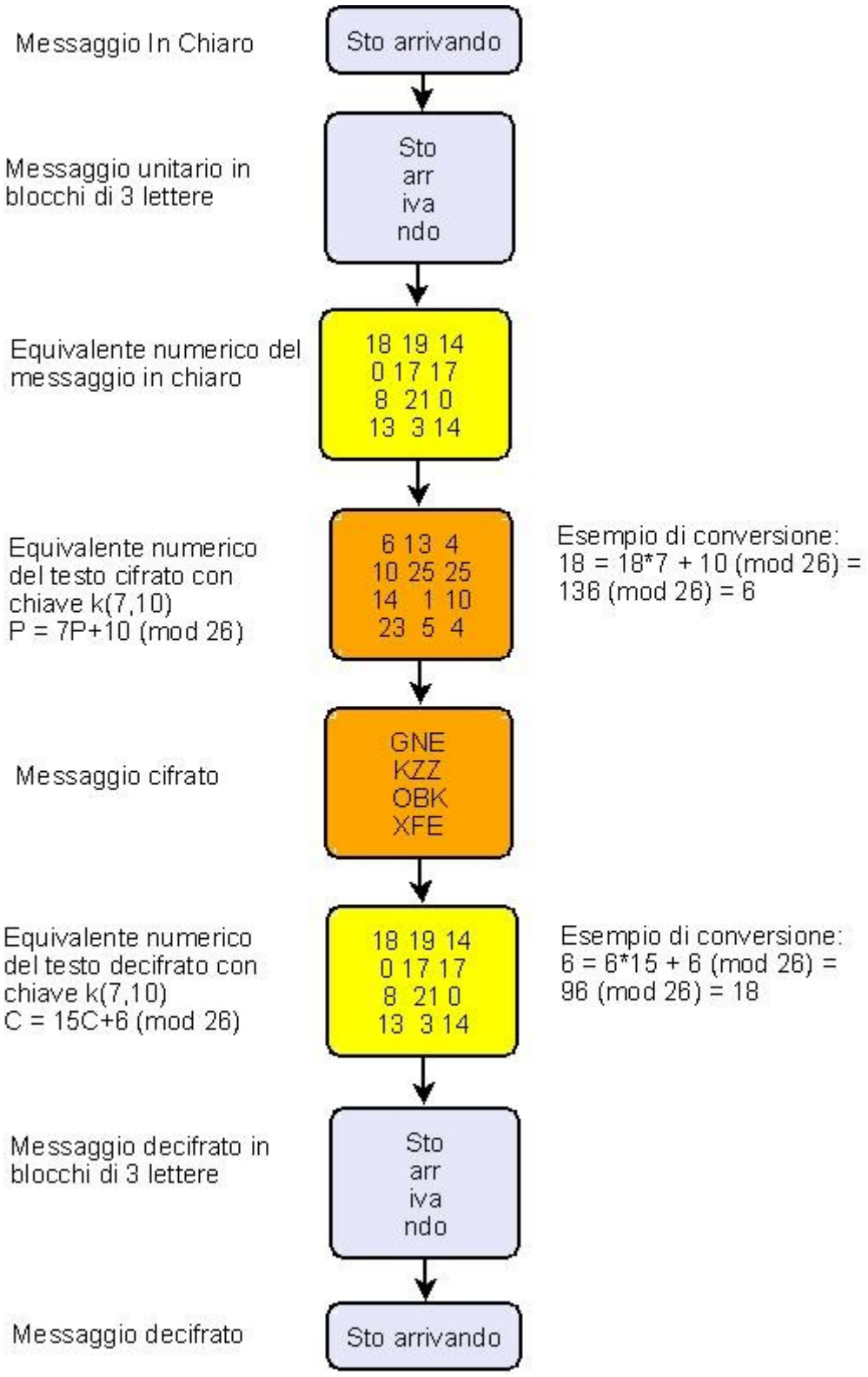
$\forall\, n \geq 1$, si indica con $\varphi(n)$ il numero di interi positivi minori di n e primi con quest'ultimi. La funzione $\varphi : N \rightarrow N$, prende il nome di funzione di Eulero.

Quindi ritornando alla nostra argomentazione sui cifrari affini, avremo che:

$$| U(Z_{26}) | = \varphi(26) = | \{1 \leq a \leq 25 \mid MCD(a,b) = 1\} | = 12$$

e dunque ci sono $12 \cdot 26 = 312$ cifrari affini, di cui uno, quello con $k=(1,0)$ è banale.

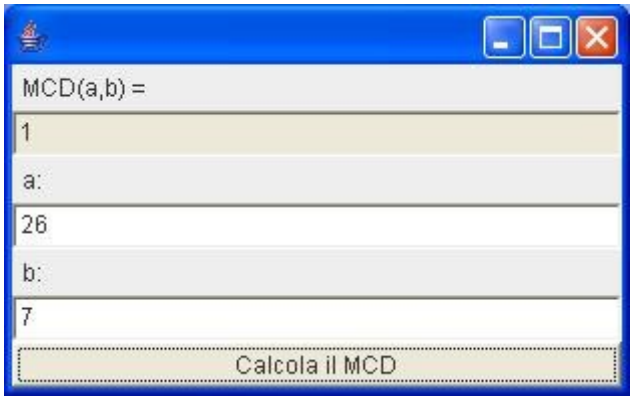
Di seguito viene proposto lo schema logico, di tale tipo di cifrario, con chiave di cifratura $k=(7,10)$, ovvero $a=7$, $b=10$ ed $N=26$, ove $C_k = (aP + b) \bmod N = (7P+10) \bmod 26$ e $D_k = C_k^{-1} = a'(C - b) \bmod N = 15(C-10) \bmod 26 = 15C + 6 \bmod 26$.



Analizziamo passo dopo passo i vari procedimenti algebrici coinvolti nel cifrario sopra proposto:

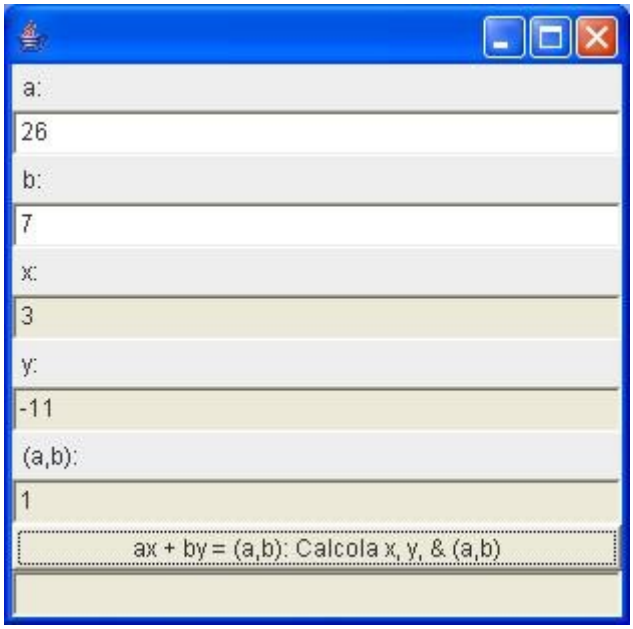
1. Calcoliamo il *MCD(26,7)* con il metodo di Euclide :
   - $26 = 7 \cdot 3 + 5$
   - $7 = 5 \cdot 1 + 2$
   - $5 = 2 \cdot 2 + \mathbf{1}$

per cui possiamo asserire che $a=7$ è relativamente primo con $N=26$ quindi la congruenza $7x \equiv 1 \pmod{26}$ ha un'unica soluzione $a' \pmod{26}$. Tale valore a' ci permetterà di definire con certezza la funzione inversa di decifratura D_k.
Si noti che il risultato MCD(26,7) viene confermato anche dalla classe *TestGCD.java*



2. Calcoliamo l'identità di Bézout $MCD(26,7) = 7x + 26y = (1,0)x + (0,1)\ y$ mediante sostituzione dei coefficienti $a=7=(1,0)$ e $b=26=(0,1)$ come combinazione lineare e sostituendo a ritroso i resti ottenuti al precedente punto 1:
   - $r_1 = 5 = 26 - 7 \cdot 3 = (0,1)+(1,0)(-3) = (0,1+(-3,0) = (-3,1)$
   - $r_2 = 2 = 7 - 5 \cdot 1 = (1,0)+(-3,1)(-1) = (1,0)+(3,-1) = (4,-1)$
   - $r_3 = 1 = 5 - 2 \cdot 2 = (-3,1)+(4,-1)(-2) = (-3,1)+(-8,2) =$ ***(-11,3)***

   infatti $1 = 7(-11) + 26(3) = -77 + 78$. Tale risultato MCD(26,7) viene confermato anche dalla classe *TestDiophantine.java*



   quinsi $a' = -11 \equiv 15 \pmod{26}$ e la funzione di decifratura è $D_{(7,10)} = C_{(7,10)}^{-1} = a'(C - b) \bmod N = 15(C-10) \bmod 26 = (15C - 150) \bmod 26 = 15C + 6 \bmod 26$

Questi passaggi sono stati tradotti nei seguenti metodi che caratterizzano la classe *Ciphers.java* la quale conterrà tutti cifrari esposti in questo documento:

```
package cifrari;

import java.math.*;
```

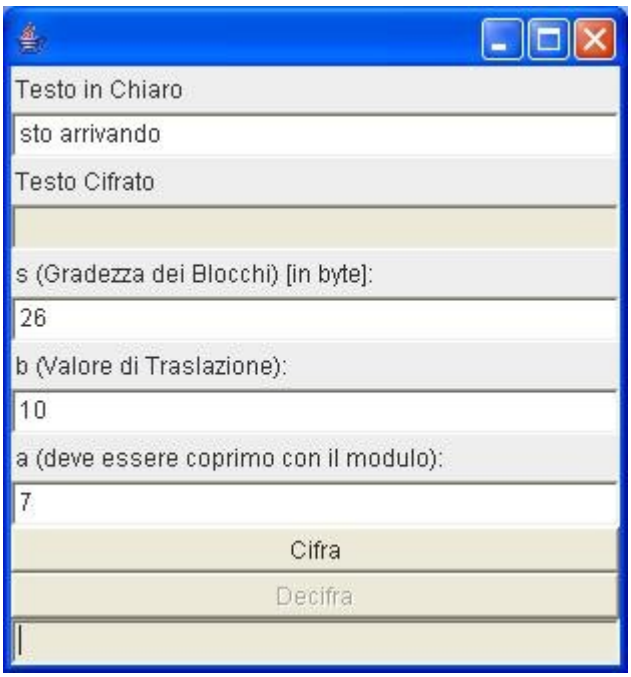
```
import algebra.BigIntegerMath;

/** Scrivere blocchi cifrati risulta difficoltoso, non tanto perchè l'operazione
 * di criptaggio risulta difficoltosa, ma perchè bisogna effettuare operazioni
di
 * impacchettamento e "spacchettamento" dei blocchi che compongono i singoli
messaggi.
 * Al fine di realizzare ciò, viene fornita la seguente classe; questi contiene
tutto quello
 * che encessi ta per l'impacchettamento e metodi sia per il criptaggio che
decriptaggio
 * mediante diverse trasformazioni.
 *  La prima tecnica sarà quella della trasformazione di un blocco con cifrari
affini.
 *
 * */

public class Ciphers {

      /**Implementa la suddivisione in blocco di un messaggio.
       * @param byte[] msg, rappresenta il messaggio;
       * @param int blockSize, rappresenta la lunghezza del pacchetto.
       * Determina il numero di blocchi utilizzando lo schema PKCS#5, ovvero:
       * Supponiamo che la grandezza del blocco sia di 64 bytes ed il messaggio
sia di 125 bytes.
       * In condizioni normali avremmo un blocco completo da 64 bytes, più uno
       * incompleto composto da soli 61 byte, facendo rimanere scoperi 3 bytes.
       * Per far fronte a questo inconveniente aggiungeremo in coda tre volte il
numero 3
       * (in binario ovviamente).
       * Es.
       *      Messaggio      Padding
       *      ... ... ... 00000011 00000011 00000011
       *
       * Con questa tecnica il messaggio risulta essere suddiviso perfettamente
in blocchi.
       * Inoltre ciò significa che ogni cifrario che usa tale metodo risulta
essre efficate
       * limitatamente ad un numero di blocchi di grandezza massima di 255
bytes.
       * @return un valore byte[], rappresentate il messaggio impacchettato.
       * */
      private static byte[] pad(byte[] msg,int blockSize) {
            //Verifica che il messaggio sia adeguato per lo schema PKCS#5
            if (blockSize<1||blockSize>255)
                  throw new IllegalArgumentException("La grandezza dei blocchi
deve essere compresa tra 1 e 255.");
            //Padding del messaggio
            int numberToPad=blockSize-msg.length%blockSize;
            byte[] paddedMsg=new byte[msg.length+numberToPad];
            System.arraycopy(msg,0,paddedMsg,0,msg.length);
            for (int i=msg.length;i<paddedMsg.length;i++)
                  paddedMsg[i]=(byte)numberToPad;

            return paddedMsg;
      }//pad

      /**Questo metodo prende in messaggio sottoposto a padding,
       * e lo converte in un array midimensionale di byte.
       *    Ogni riga in questa matrice è un blocco.
       * Il metodo di decriptaggio lavorerarà con questi due array.
```

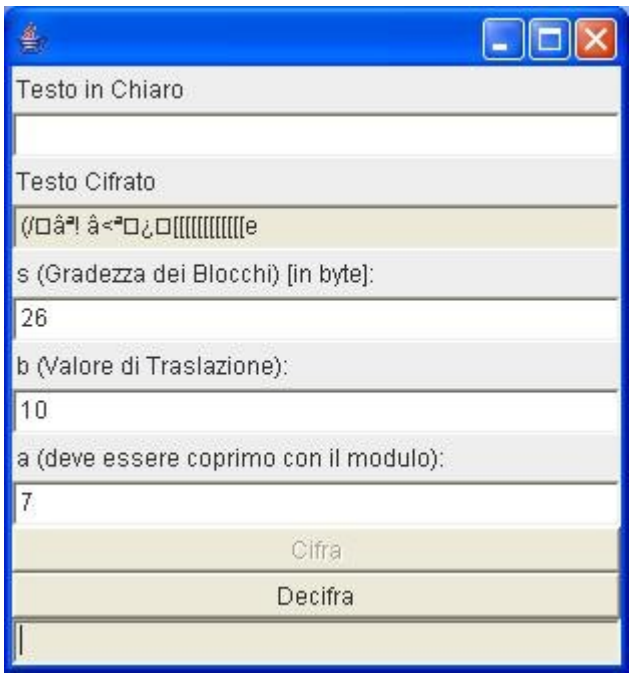
```
       * @param byte[] msg, rappresenta il messaggio già sottoposto alla tecnica
del padding;
       * @param int blockSize, rappresenta la lunghezza del pacchetto.
       * @return una matrice byte[][], rappresentate il messaggio in due
blocchi.
       * */
      private static byte[][] block(byte[] msg,int blockSize) {
            //Crea un array di bytes 2D sottoposto alla tecnica del padding
            int numberOfBlocks=msg.length/blockSize;
            byte[][] ba=new byte[numberOfBlocks][blockSize];
            for (int i=0;i<numberOfBlocks;i++)
                  for (int j=0;j<blockSize;j++)
                        ba[i][j]=msg[i*blockSize+j];

            return ba;
      }//block

      /**Questo metodo spacchetta il messaggio; Ciò avviene dopo l'operazione di
criptaggio o decriptaggio.
       * Questi riceve in input un array 2D di bytes e lo riconverte in un array
lineare di bytes.
       * Nell'applicare tale metodo bisogna avere accortenza nel verificare che
       * la trasformazione di cifratura o decifratura possa produrre un numero
intero inferiore alla
       * dimensione del blocco stesso.
       * In tal caso, si provvederà a riempire la parte finale del blocco.
       * @param byte[][] ba, rappresenta una matrice nella quale è memorizzato
il messaggio in due blocchi.
       * @param int blockSize, rappresenta la lunghezza del pacchetto.
       * @return un array byte[], dove memorizzare il codice decrifrato
       * */
      private static byte[] unBlock(byte[][] ba,int blockSize) {

            //Crea un array byte[], dove memorizzare il codice decrifrato
            byte[] m2=new byte[ba.length*blockSize];
            //Pone i blocchi in un array 1D
            for (int i=0;i<ba.length;i++) {
                  int j=blockSize-1;
                  int k=ba[i].length-1;
                  while (k>=0) {
                        m2[i*blockSize+j]=ba[i][k];
                        k--;
                        j--;
                  }//while
            }//for

            return m2;
      }//unBlock

      /**
       * Questo metodo rimuove il padding dal messaggio.
       * Esamina il valore numerico dell'ultimo blocco ed in base
       * a tale valore ne elimina una quantità pari di blocchi
       * aggiunti durante la fase di padding.
       *
       * @param byte[] msg, rappresenta il messaggio con padding.
       * @param int blockSize, rappresenta la lunghezza del pacchetto.
       * @return un array byte[], sprovvisto di blocchi di padding.
       * */
```

```
	private static byte[] unPad(byte[] msg,int blockSize) {
		/*Determina la quantitù di blocchi, testando il valore memorizzato
		nell'ultimo blocco*/
		int numberOfPads=(msg[msg.length-1]+256)%256;
		//Elimina i blocchi di padding
		byte[] answer=new byte[msg.length-numberOfPads];
		System.arraycopy(msg,0,answer,0,answer.length);
		return answer;
	}//unPad

	/**
	 * Metodo per criptare mediante cifrari affini.
	 * Accetta un messaggio, la grandezza del blocco ed il valore dei
coefficienti
	 * <i>a</i> e <i>b</i> della trasformazione C = aP + b.
	 * @param byte[] msg, rappresenta il messaggio con padding.
	 * @param int blockSize, rappresenta la lunghezza del pacchetto.
	 * @param BigInteger a, rappresenta il coefficiente moltiplicativo del
testo in chiro
	 * @param BigInteger b, rappresenta il coefficiente additivo del testo in
chiaro
	 * @exception IllegalArgumentException, se il parametro <i>a</i> non è
coprimo con il modulo
	 * @return un array byte[], un messaggo criptato con cifrari affini.
	 * */
	public static byte[] affineEncipher(byte[] msg,int blockSize,BigInteger
a,BigInteger b) {
		//Calcola il modulo
		BigInteger modulus=BigInteger.valueOf(2).pow(8*blockSize);
		//verifica se 'a' è coprimo con il modulo
		if (!(modulus.gcd(a).equals(BigIntegerMath.ONE)))
			throw new IllegalArgumentException("I coefficienti di
criptaggio non sono coprimi con il modulo.");
		byte ba[][]=block(pad(msg,blockSize),blockSize);
		//Inizio criptaggio
		for (int i=0;i<ba.length;i++)
			ba[i]=getBytes(a.multiply(new
BigInteger(ba[i])).add(b).mod(modulus));
		return unBlock(ba,blockSize);
	}//affineEncipher

	/**
	 * Metodo per decrifrare mediante cifrari affini.
	 * Accetta un messaggio, la grandezza del blocco ed il valore dei
coefficienti
	 * <i>a</i> e <i>b</i> della trasformazione C = aP + b.
	 * @param byte[] msg, rappresenta il messaggio con padding.
	 * @param int blockSize, rappresenta la lunghezza del pacchetto.
	 * @param BigInteger a, rappresenta il coefficiente moltiplicativo del
testo in chiro
	 * @param BigInteger b, rappresenta il coefficiente additivo del testo in
chiaro
	 * @exception IllegalArgumentException, se il parametro <i>a</i> non è
coprimo con il modulo
	 * @return un array byte[], un testo in chiaro.
	 * */
	public static byte[] affineDecipher(byte[] msg,int blockSize,BigInteger
a,BigInteger b) {
		//Calcola il modulo
		BigInteger modulus=BigInteger.valueOf(2).pow(8*blockSize);
		//verifica se 'a' è coprimo con il modulo
```

```
            if (!(modulus.gcd(a).equals(BigIntegerMath.ONE)))
                  throw new IllegalArgumentException("I coefficienti di
criptaggio non sono coprimi con il modulo.");
            //Calcola l'inverso del coefficiente 'a'
            BigInteger ainv=a.modInverse(modulus);
            byte[][] ba=block(msg,blockSize);
            //Inizio decriptaggio
            for (int i=0;i<ba.length;i++)
                  ba[i]=getBytes(BigIntegerMath.lnr(ainv.multiply(new
BigInteger(ba[i]).subtract(b)),modulus));
            //Trasforma i blocchi in un array a 1De rimuove il padding da
quest'ultimi
            return unPad(unBlock(ba,blockSize),blockSize);
      }//affineDecipher

      /**
       * Questo metodo, oltre che a restituire la rappresentazione binaria di un
BigInteger,
       * restituisce anche il suo segno nella prima posizione.
       * Mediante quest'ultimi risulta possibile la gestione dei pacchetti da
accodare nella parte
       * inferiore della lista.
       * */
      private static byte[] getBytes(BigInteger big) {
            byte[] bigBytes=big.toByteArray();
            if (big.bitLength()%8!=0)
                  return bigBytes;
            else {
                  byte[] smallerBytes=new byte[big.bitLength()/8];
                  System.arraycopy(bigBytes,1,smallerBytes,
0,smallerBytes.length);
                  return smallerBytes;
            }
      }//getBytes

}//Ciphers
```

la cui implementazione grafica sullo standard ASCII è stata definita nella classe *TestBlockAffineCipher.java* :






```
package cifrari;

import java.math.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

import algebra.TestGCD;

public class TestBlockAffineCipher extends JFrame implements ActionListener {

   BigInteger shift=null;
   BigInteger multiplier=null;
   int blockSize=0;
   byte[] msgArray=null;
   byte[] encmsgArray=null;

   Label Label1=new Label("Testo in Chiaro");
   TextField msg=new TextField(40);
   Label Label2=new Label("Testo Cifrato");
   TextField encmsg=new TextField(40);
   Label shiftLabel=new Label("b (Valore di Traslazione):");
   TextField entryShiftValue=new TextField(40);
   Label blockLabel=new Label("s (Gradezza dei Blocchi) [in byte]:");
   TextField entryBlockValue=new TextField(40);
   Label multLabel=new Label("a (deve essere coprimo con il modulo):");
   TextField entryMultValue=new TextField(40);
   Button encipherButton=new Button("Cifra");
   Button decipherButton=new Button("Decifra");
   TextField outmsg=new TextField("");

   public TestBlockAffineCipher() {

         Panel p1=new Panel();

         p1.setLayout(new GridLayout(13,1));
         p1.add(Label1);
         p1.add(msg);
```

```
        p1.add(Label2);
        p1.add(encmsg);
        encmsg.setEditable(false);
        p1.add(blockLabel);
        p1.add(entryBlockValue);
        p1.add(shiftLabel);
        p1.add(entryShiftValue);
        p1.add(multLabel);
        p1.add(entryMultValue);
        p1.add(encipherButton);
        encipherButton.addActionListener(this);
        p1.add(decipherButton);
        decipherButton.addActionListener(this);
        decipherButton.setEnabled(false);
        p1.add(outmsg);
        outmsg.setEditable(false);
      add(p1);
      pack();
      show();
  }

    public void actionPerformed(ActionEvent e) {
       if (e.getSource()==encipherButton) {
          outmsg.setText("");
          try {
             blockSize=Integer.parseInt(entryBlockValue.getText());
          } catch (NumberFormatException nfe) {
             blockSize=1;
             entryBlockValue.setText("1");
      }
          try {
             shift=new BigInteger(entryShiftValue.getText());
          } catch (NumberFormatException nfe) {
             shift=BigInteger.valueOf(0);
             entryShiftValue.setText("0");
      }
          try {
             multiplier=new BigInteger(entryMultValue.getText());
          } catch (NumberFormatException nfe) {
             multiplier=BigInteger.valueOf(1);
             entryMultValue.setText("1");
      }
          msgArray=msg.getText().getBytes();
          try {
             encmsgArray=Ciphers.affineEncipher(msgArray,blockSize,multiplier,shi
ft);
          } catch (IllegalArgumentException iae) {
             outmsg.setText(iae.toString());
      }
          try {
             encmsg.setText(new String(encmsgArray));
          } catch (NullPointerException npe) {
             outmsg.setText(npe.toString());
      }
          msg.setText("");
          encipherButton.setEnabled(false);
          decipherButton.setEnabled(true);
       } else if (e.getSource()==decipherButton) {
          outmsg.setText("");
```

```
            try {
               msgArray=Ciphers.affineDecipher(encmsgArray,blockSize,multiplier,shi
ft);
            } catch (IllegalArgumentException iae) {
               outmsg.setText(iae.toString());
        }
            msg.setText(new String(msgArray));
            encmsg.setText("");
            decipherButton.setEnabled(false);
            encipherButton.setEnabled(true);
      }
    }

      public static void main(String[] arg){
               new TestBlockAffineCipher();
         }//main

}
```

I principali algoritmi di crittografia a chiave segreta

Di seguito vi proponiamo alcuni tra i più famosi algoritmi a chiave simmetrica:

- **DES (Data Encryption Standard)** : fu sviluppato da IBM nel 1970 e definito dal governo degli Stati Uniti come standard ufficiale nel1977. La dimensione dei blocchi DES è di 64 bit e usa una chiave a 56 bit durante la cifratura.
- **Triple-DES** : è l'evoluzione del precedente ed è basato su un utilizzo del cifrario DES ripetuto, con due o tre chiavi di 56 bit ciascuna. L'algoritmo Triple-DES può essere di tre tipi: DES-EEE3 (cifra-cifra-cifra), DES-EDE3 (cifra-decifra-cifra) e DES-EEE2 (cifra-cifra-cifra). Il DES-EEE3 usa tre differenti chiavi durante la cifratura. Il DES-EDE3 usa tre differenti chiavi ugualmente, ma nella sequenza cifra-decifra-cifra. Il DES-EEE2 è come il DES-EEE3, ma usa solo due chiavi diverse.
- **IDEA (Intenational Data Encryption Algorithm)** : ha una chiave lunga 128 bit.
- **RC5** : Sviluppato da Ronald Rivest per la RSA Data Security, è un algoritmo parametrico con lunghezza di chiave variabile. La dimensione della chiave può essere da 0 fino a 2048 bit.
- **Blowfish** : ideato nel 1993 da bruce Schneier. È stato sviluppato come algoritmo di cifratura: veloce, compatto, semplice da implementare e sicuro con chiavi di dimensioni variabili fino a 448 bit. È un algoritmo non protetto da copyright, utilizzato in molti sistemi Open Source.
- **Rijndael** : sviluppato da Joan Daemen e Vincent Rijmen, ha vinto la selezione per l'Advanced Encryption Standard (AES) nel 2000. Ufficialmente il Rijndael è diventato lo standard per la cifratura del XXI secolo. Il cifrario utilizza chiavi di lunghezza variabile 128, 192, 256 bit.

La crittografia a chiave pubblica o asimmetrica

La crittografia a chiave pubblica fu introdotta nel 1976 a Whitfield Diffie e Martin Hellman. Il suo funzionamento è semplice solo in apparenza, poiché in realtà è basato su modelli matematici abbastanza articolati, i quali comunque verranno discussi in seguito.

Nella crittografia asimmetrica ogni individuo che intende comunicare dispone di una coppia di

chiavi di cui una *pubblica* e l'altra *privata*. La prima è pubblicata ed accessibile a tutti, in modo che chiunque la voglia usare lo possa fare. La chiave privata è custodita con la massima cautela dal proprietario e non va scambiata con nessuno. Questo sistema elimina quindi il problema della comunicazione della chiave segreta.

Nella crittografia asimmetrica, ciò che viene crittografato con la chiave pubblica, operazione che può essere fatta da chiunque, può essere decifrato solo con la chiave privata corrispondente, operazione che può essere fatta solo dal proprietario della della chiave. È anche possibile il contrario, ovvero solo il proprietario può cifrare con la chiave privata e tutti gli altri possono decifrare con quella pubblica[6].

Come vedremo successivamente con maggiore dettaglio nei paragrafi a seguire, questa asimmetria permette di soddisfare tutti i requisiti nello scambio di informazioni, in quanto:

- la *riservatezza* viene garantita se il mittente cifra il messaggio con la chiave pubblica del destinatario: solo quest'ultimo potrà decifrarlo con la propria chiave privata;
- l'*integrità* viene garantita, anche in modo separato dalla riservatezza con l'uso di un'impronta digitale denominata *hash*, cifrata dal mittente con la propria chiave privata e decifrata all'arrivo con la corrispondente chiave pubblica : tutti possono verificare che il messaggio non sia stato alterato.

L'*autenticità* ed il *non ripudio* sono assicurati con lo stesso procedimento.

Oltre a questi requisiti, la crittografia asimmetrica offre altri due vantaggi. Permette di avere la certezza del momento in cui un messaggio sia stato inviato, banalmente applicando la tecnica dell'hash alla data di spedizione dello stesso. Infine può garantire non solo l'autenticità del mittente, ma anche l'identità dello stesso, mediante l'uso dei famosi *certificati a chiave pubblica*, i quali anche loro verranno trattati nel dettaglio in seguito.

Il metodo della crittografia asimmetrica si basa su due concetti fondamentali:

- un messaggio cifrato con una delle due chiavi può essere decifrato solo con l'altra;
- non è possibile, o meglio è computazionale sconveniente, ricavare una chiave dall'altra. Anche tale caratteristica matematica, verrà dimostrata in seguito.

Descriviamo ora le diverse modalità di applicazione della crittografia asimmetrica.

La riservatezza dei messaggi

Volendo comunicare in modo riservato con qualcuno, non dobbiamo far altro che procurarci la sua chiave pubblica depositata opportunamente in un idoneo registro, ovviamente anch'esso pubblico, per poi cifrare con essa il messaggio da trasmettere. Il nostro interlocutore quando riceve il messaggio cifrato può decifrarlo con la propria chiave privata. Tornando ai nostri amici Alice e Bob, nella figura sottostante viene mostrato il flusso appena descritto.

6 Vedremo che quest'ultima tecnica la si applica alla firma digitale

L'autenticità del mittente

Il precedente metodo per la riservatezza dei dati garantisce che solo Bob possa leggere il messaggio trasmesso da Alice, ma, come precedentemente anticipato, potrebbe comunque succedere che qualcuno poco poco leale si sostituisca a quest'ultima. Nasce quindi l'esigenza di assicurare l'autenticità del mittente. Per fare questo basta modificare il flusso delle operazioni di crittografia asimmetrica nel modo illustrato nella figura sottostante.

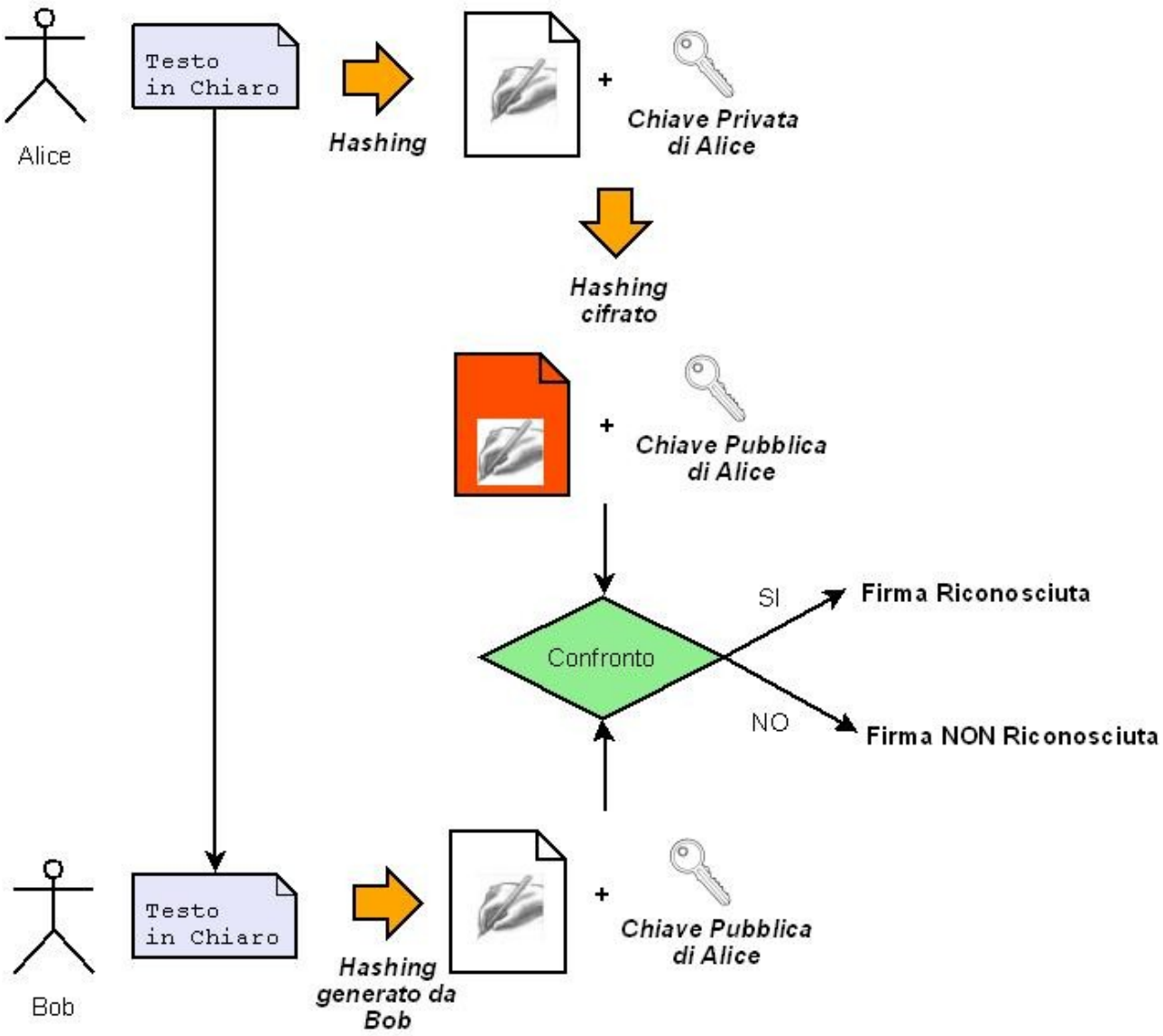
Alice cifrando il testo del messaggio con la propria chiave privata, ne attesta l'autenticità : praticamente è come se firmasse il messaggio di proprio pugno. Bob ha la garanzia che il mittente del messaggio sia veramente Alice, ma purtroppo, chiunque potrebbe leggere il messaggio poiché la chiave di decifratura è pubblica : viene quindi a mancare la riservatezza del messaggio.

Per assicurare contemporaneamente sia la riservatezza del messaggio che l'autenticità del mittente, basta eseguire i due processi appena descritti, in sequenza!

L'autenticità del mittente e l'integrità del messaggio

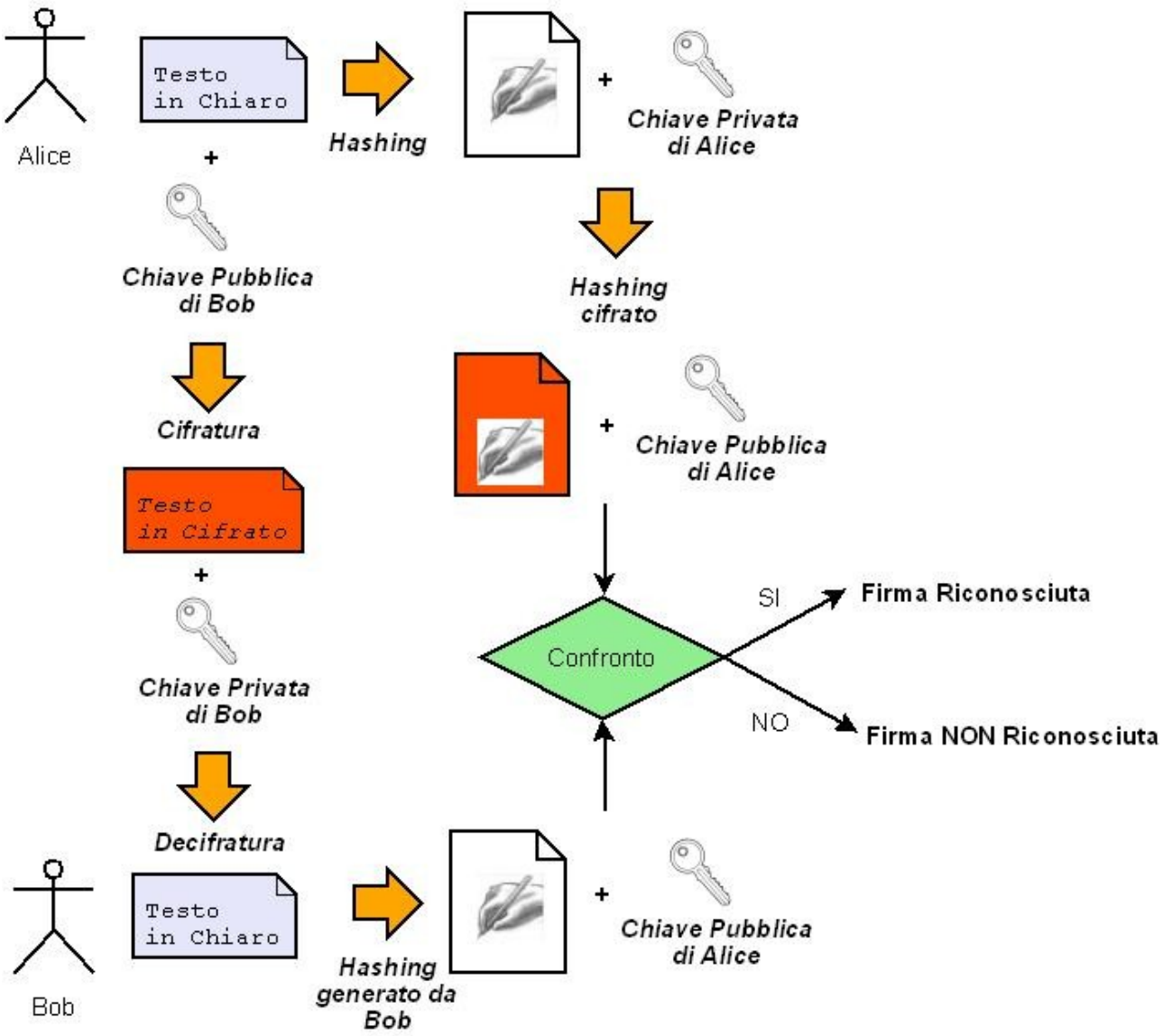
Per garantire contemporaneamente l'autenticità del mittente e l'integrità del messaggio, risulta necessario introdurre una nuova tecnica denominata *hashing* che consiste nella generazione di un'impronta digitale del messaggio, detto anche *digest*. Quest'impronta è sostanzialmente una stringa di lunghezza fissa ricavata dal testo del messaggio secondo algoritmi di hash. Essa consente, tramite una comparazione successiva, di verificare se il messaggio pervenuto dal destinatario sia corrispondente all'originale, cioè integro.

Alice deve calcolare un hash del testo del messaggio da inviare, ottenere la corrispondente impronta digitale, cifrarla con la propria chiave privata ed inviare il tutto, ovvero messaggio in chiaro ed impronta, a Bob. Bob, al ricevimento del pacchetto, ovvero messaggio in chiaro ed impronta cifrata, deve generare per conto proprio l'hash del messaggio in chiaro, decifrare l'impronta cifrata e comparare i due hash : quello ricevuto da Alice e quello generato da lui stesso. Se i due hash corrispondono, il messaggio è arrivato integro ed è inoltre garantita l'autenticità del mittente. La figura sottostante mostra il flusso delle operazioni appena descritte.



L'autenticità del mittente, la riservatezza e l'integrità del messaggio

Per assicurare contemporaneamente questi tre requisiti, basta usare le due coppie di chiavi del mittente e del destinatario. Alice deve quindi cifrare il messaggio con la chiave pubblica di Bob ed inoltre cifrare l'hash con la propria chiave privata. In questo modo otteniamo l'integrità e la riservatezza del messaggio, oltre all'autenticità del mittente. La figura sottostante mostra il flusso delle operazioni appena descritte.



Cifrari a chiave pubblica

Come è stato illustrato nei paragrafi precedenti, un cifrario a chiave pubblica è un cifrario che permette di divulgare il metodo ed anche la chiave di cifratura, da cui il nome di *cifrario a chiave pubblica*, senza per questo rilevare contestualmente il modo di decifrare. Dal punto di vista matematico, tale sistema è realizzabile grazie alla difficoltà nel calcolare in un tempo *ragionevolmente breve*[7] la trasformazione di decifratura, inversa di quella di cifratura, infatti è necessario essere in possesso di altre informazioni oltre quelle rese pubbliche. Tali informazioni però sono tenute segrete e senza di esse la complessità di calcolo delle decifratura è tale da renderla implausibile[8]: in sostanza, dalla c.d. *Ipotesi di Diffie-Hellman*, si congettura che la complessità del calcolo delle chiavi di cifratura e decifratura, anche conoscendone una sola, sia uguale a quella della ricerca dei logaritmi discreti.

Nel paragrafo a seguire verrà illustrato uno dei principali algoritmi nel quale si implementa tale criterio di crittoanalisi, ovvero l'algoritmo RSA.

7 Tempo polinomiale. Si veda il paragrafo del calcolo del logaritmo discreto.
8 esponenziale

Il sistema RSA

Il sistema RSA fu sviluppato nel 1977 da Ron Rivest, Adi Shamir e Leonard Adleman. Le iniziali dei loro cognomi hanno dato origine al nome dell'algoritmo. La sicurezza del sistema è data appunto dalla difficoltà di fattorizzare un prodotto di due numeri primi di dimensione elevate. La lunghezza delle chiavi è variabile : 512, 1024, 2048, 4096 bit e oltre. Essenzialmente, in tale sistema, ogni utente pubblica la propria chiave di cifratura per permettere a chiunque voglia comunicare con lui segretamente.

Dal punti di vista matematico, le chiavi di cifratura possono essere come una coppia di interi positivi *(n , e)*, tali che:

- *n* è prodotto di due interi primi grandi *p* e *q* che conosciamo solo noi;
- *e* deve essere relativamente primo con $\varphi(n) = (p-1)(q-1)$, ossia $MCD(e , \varphi(n))=1$, cioè $MCD(e, p-1)=MCD(e, q-1)=1$.

Tale coppia di interi *(n , e)* viene pubblicata in un elenco di dominio pubblico.

Quindi, in generale, ogni utente *U* che voglia servirsi di questo sistema dovrà procedere analogamente, consegnando cioè una coppia di interi (n_u , e_u) verificanti le stesse condizioni:

- n_u deve essere prodotto di due interi primi grandi p_u e q_u che deve tener segreti;
- e_u deve essere relativamente primo con $\varphi(n_u) = (p_u-1)(q_u-1)$, ossia $MCD(e , \varphi(n_u))=1$, cioè $MCD(e, p_u-1)=MCD(e, q_u-1)=1$.

Si sottolinea ancora una volta che la coppia (n_u , e_u) è di dominio pubblico, ossia un qualunque utente che lo desideri può consultarla, mentre *non è di dominio pubblico la fattorizzazione di* n_u *, nota solamente ad U*.

Per continuare la descrizione del sistema RSA seguiamo il seguente esempio: ipotizziamo che o pubblico delle chiavi del nostro sistema RSA sia così composto:

ID_Utente	n	e
A	77	13
B	1003	3
C	247	5
D	703	7

I numeri sono stati scelti dall'utente secondo le condizioni richieste, perché:

- $n_A = 77 = p_A \times q_A = 7 \times 11 \mid MCD(e_A, p_A-1)=MCD(13,6)=MCD(e, q_A-1)=MCD(13,10)=1$
- $n_B = 1003 = p_B \times q_B = 17 \times 59 \mid MCD(e_B, p_B-1)=MCD(3,16)=MCD(e, q_B-1)=MCD(3,58)=1$
- $n_C = 247 = p_C \times q_C = 13 \times 19 \mid MCD(e_C, p_C-1)=MCD(5,12)=MCD(e, q_C-1)=MCD(5,18)=1$
- $n_D = 703 = p_D \times q_D = 19 \times 37 \mid MCD(e_D, p_D-1)=MCD(7,18)=MCD(e, q_D-1)=MCD(7,36)=1$

Osserviamo, che nell'esempio che stiamo considerando abbiamo scelto dei numeri piccoli per i quali è facile trovare i due numeri primi p_U e q_U tali che $n_U = p_U \times q_U$. In generale l'utente *U*, per essere sicuro, utilizzerà un intero n_U che sia prodotto di due primi di circa *100* cifre decimali, e quindi il numero n_U loro prodotto, sarà pubblicato, avrà circa *200* cifre.

Aprendo una breve parentesi implementativa, chi volesse ricavare un numero del genere può utilizzare il metodo *getPrimeNbit(bitlength)* della classe *BigIntegerMath.java*, il quale restituisce un numero intero primo lungo *n* bit. Ad esempio moltiplicando due primi da *512* bit potremmo ottenere

il seguente numero:

n =
72010606896114604614554441303619089214666495965659789569269806431816420004682594
61156452359986917097279926004190801669254018565578186261884390173745989780100459
81925767569715323988070634125766802328150792798337990331865196502330237888274540
29905877410204164193045059148313322760193020576875546323388315248 09

i cui fattori si possono ottenere solo dopo alcuni mesi, utilizzando ad esempio il metodo *p-1 di Pollard*, illustrato nel precedente capitolo.

Invio di un messaggio cifrato con il sistema RSA

Ipotizziamo che l'utente *A*, voglia inviare con il sistema RSA la parola *'algebra'* all'utente *B*. Per fare ciò l'utente *A* dovrà procedere come segue:

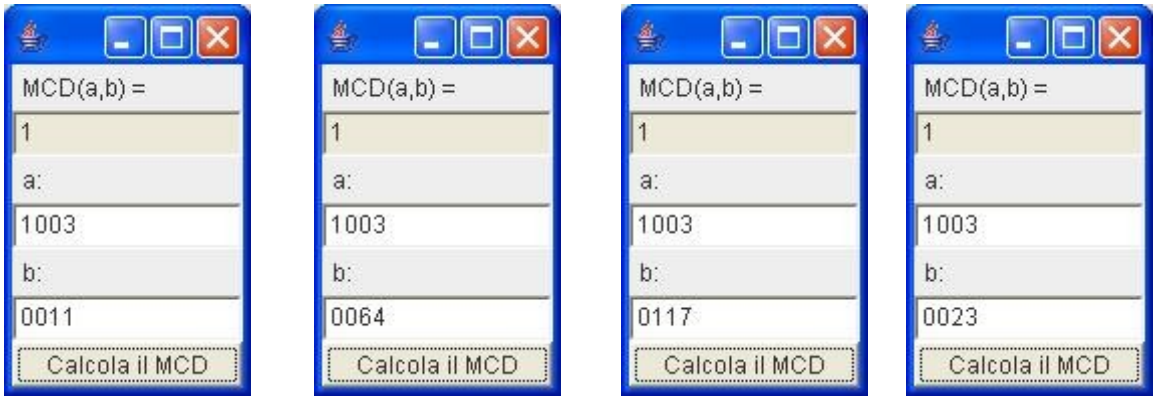
1. Per prima cosa trasformare tutte le lettere del messaggio in un intero in base all'equivalenza numerica a due cifre, secondo il criterio precedentemente illustrato nel paragrafo introduttivo all'impostazione matematica di un crittosistema. Quindi la sequenza di numeri corrispondenti al nostro messaggio sarà :

a	*l*	*g*	*e*	*b*	*r*	*a*
00	*11*	*06*	*04*	*01*	*17*	*00*

2. Successivamente l'utente *A* consulterà l'elenco pubblico delle chiavi RSA, dal quale ricaverà la coppia (n_B , e_B) associata ovviamente all'utente *B*. Per il momento serve solo $n_B = 1003$.
3. A questo punto deve suddividere il messaggio da inviare in messaggi unitari *in modo che l'intero associato ad ogni messaggio unitario sia minore di* $n_B = 1003$ *e relativamente primo con 1003*. Osserviamo che dividendo il messaggio in blocchi di due lettere, ossia *digrafi*, al modo seguente:

al | ge | br | ax

0011 | 0604 | 0117 | 0023



in modo tale che otteniamo tutti numeri minori e relativamente primi con *1003*. Osserviamo che per verificare il *MCD* tra *1003* ed i predetti numeri, l'utente *A* potrebbe utilizzare l'algoritmo euclideo delle divisioni successive, implementato nel metodo *gcd(first, second)* della classe BigIntgerMath.java e successivamente testato con la classe *TestGCD.java*, dato che non conosce la fattorizzazione di n_B in fattori primi.

Si noti ancora che l'utente *A* ha dovuto aggiungere al messaggio unitario finale la lettera *X*,

in modo tale da farlo diventare un digrafo. Se avesse pensato di dividere il messaggio in blocchi di tre lettere anziché di due, ci sarebbero dei messaggi unitari cui corrispondono degli interi maggiori di n_B :

- alg $\rightarrow$ 606 < 1003
- ebr $\rightarrow$ 40117 > 1003
- axx $\rightarrow$ 2323 > 1003

il che non avrebbe soddisfatto la nostra richiesta.
Dunque i singoli segmenti del messaggio in chiaro saranno rappresentati dai seguenti numeri:

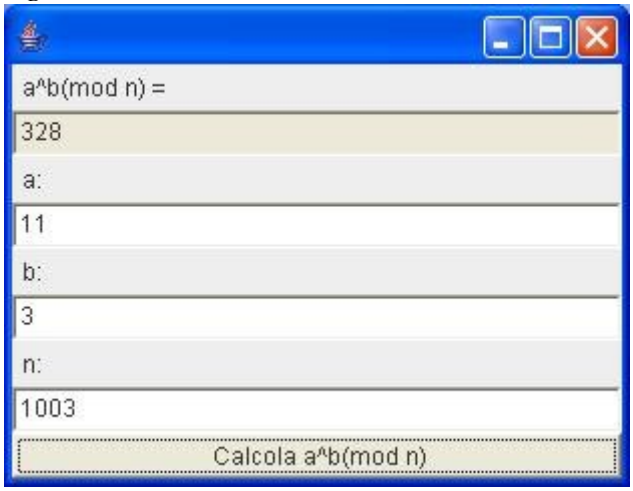
- $P_1 = 11$
- $P_2 = 604$
- $P_3 = 117$
- $P_4 = 23$

4. Ora ha inizio l'operazione di cifratura vera e propria per l'invio del messaggio all'utente *B*, in modo tale che quest'ultimi lo possa *decifrare e sia l'unica a poterlo fare in un tempo ragionevole*[9]. Per spedire il messaggio da spedire all'utente *B*, l'utente *A* eleva ogni P_i alla potenza e_B , essendo e_B il secondo elemento della coppia associata all'utente *B*.
La funzione di cifratura risulta essere quindi

$$C_B : P \rightarrow C \qquad P \rightarrow C = P^{e_B} \, mod \, n_B$$

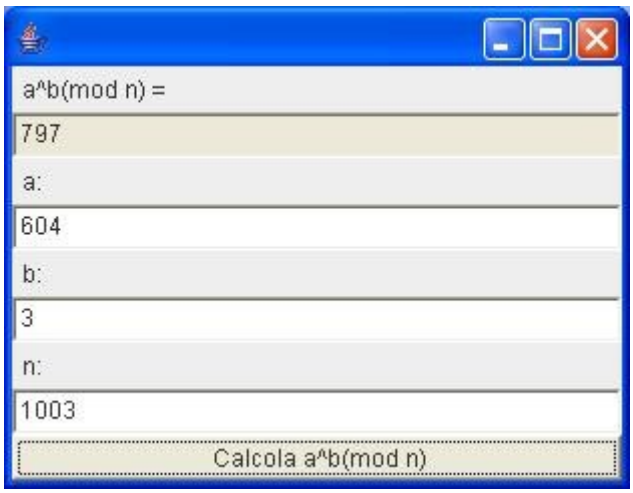
il messaggio cifrato che l'utente *A* deve mandare sarà dunque rappresentato dai seguenti numeri:

- $C_1 = P_1^{\,eB} \bmod n_B = 11^3 \bmod 1003 = 328$

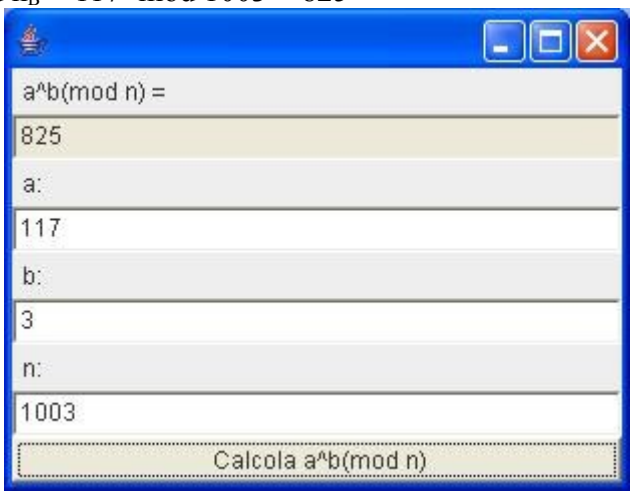


- $C_2 = P_2^{\,eB} \bmod n_B = 604^3 \bmod 1003 = 797$

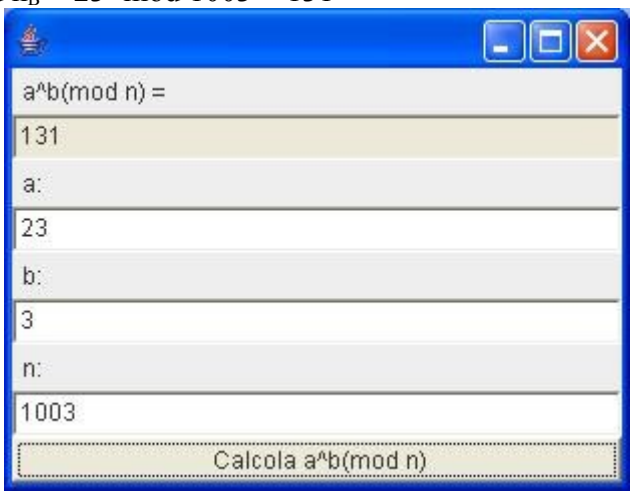
9 Tempo polinomiale



- $C_3 = P_3^{eB} \bmod n_B = 117^3 \bmod 1003 = 825$



- $C_4 = P_4^{eB} \bmod n_B = 23^3 \bmod 1003 = 131$



Si noti come tali valori siano stati calcolati direttamente con la classe *TestModPow.java*, illustrata nel capitolo precedente.

L'utente *B* riceve quindi il messaggio:

- $C_1 = 328$
- $C_2 = 797$
- $C_3 = 825$

- $C_4 = 131$

Costituito dai numeri unitari C_i , $i = 1, \ldots, 4$. A questo punto l'utente *A* ha terminato il suo compito: ha inviato all'utente *B* il messaggio cifrato. Ora sarà compito dell'utente *B* decifrarlo.

Decifratura di un messaggio cifrato con il sistema RSA

L'utente *B* ha ricevuto il messaggio:

- $C_1 = 328$
- $C_2 = 797$
- $C_3 = 825$
- $C_4 = 131$

e deve procedere alla decifratura, cioè per ogni C_i , $i = 1, \ldots, 4$, deve scoprire il messaggio originario P_i , sapendo che

$$C_i = (P_i)^{e_B} \bmod n_B$$

l'utente *B* deve quindi determinare la funzione decifrante

$$D_B : C \to P \quad tale\,che \quad D_B(E_B(P_i)) = P_i \quad \forall i$$

Come la si determina? A priori sembrerebbe che, per risolvere questo problema, l'utente *B* debba trovare un logaritmo discreto, il che, come abbiamo osservato nel capitolo precedente, è computazionalmente oneroso. Tuttavia abbiamo già detto all'inizio che, in realtà, l'utente *B* ha *un'informazione addizionale che gli consente di decifrare il messaggio ottenuto*, in maniera abbastanza agile. Vediamo di che informazione si tratta e come l'utente *B* determina innanzitutto d_B, con

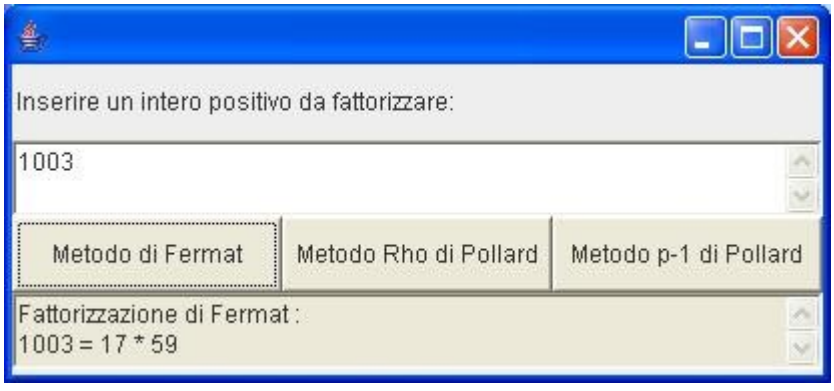
$$1 \le d_B < \varphi(n_B) = (p_B - 1)(q_B - 1)$$

tale che d_B sia soluzione della seguente congruenza:

$$e_B d_B \equiv 1 \ (mod\ \varphi(n_B))$$

tale soluzione esiste ed è unica, perché come noto $MCD(e_B, \varphi(n_B)) = 1$.

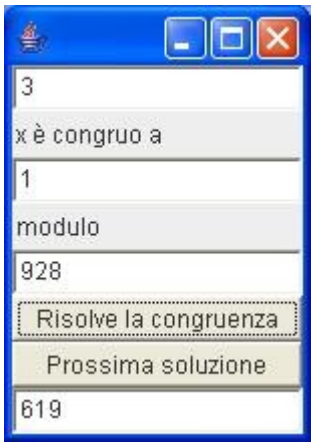
Nel caso in esame, $n_B = 1003$ e l'utente *B* conosce $\varphi(1003)$ *perché sa che* $1003 = 17 \times 59$.



Quindi sapendo anche che $\varphi(n_B) = (p_B - 1)(q_B - 1)$ ricava facilmente che $\varphi(1003) = (17-1) \times (59-1) = 16 \times 58 = 928$.

Allora la soluzione della congruenza $e_B d_B \equiv 1 \ (mod\ \varphi(n_B))$, cioè di $3x \equiv 1 \ (mod\ 928)$ è $d_B = 619$.

Da un punto di vista puramente implementativo, si noti come tale congruenza lineare, possa essere risolta mediante l'utilizzo del metodo *solveLinearCongruence(a,b,m)* della classe *BigIntegerMath.java* illustrato all'inizio del capitolo precedente, e successivamente testato con la classe *TestLinearCongruence.java*.



Per ora memorizziamo e teniamo da parte questo numero, perché, come vedremo fra un momento, l'utente *B* è l'unico che può decifrare il messaggio inviatogli, perché è l'unico che, conoscendo la fattorizzazione di *1003* è in grado di conoscere la funzione di Eulero e quindi di calcolare d_B , che è la sua *chiave segreta*. Vediamo come procede l'utente *B* per decrittare i messaggi C_i :

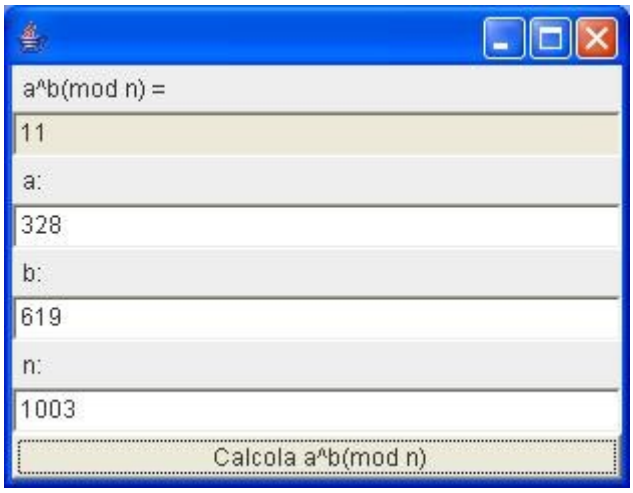
1. eleva ogni C_i a $d_B = 619$, cioè calcola: 328^{619} , 797^{619} , 825^{619} e 131^{619} , tutto ovviamente modulo *1003*.
   Ovviamente l'esponente *619* è grande, ma abbiamo già visto nel capitolo precedente come si procede in questi casi. Si scrive *619* in base *2*, cioè:
   $619 = (1001101011)_2 = 1\times2^9+ 1\times2^6+ 1\times2^5+ 1\times2^3+ 1\times2^1+1\times2^0= 512 + 64 + 32 + 8 + 2 + 1$
   allora si ha
   $C_i^{619}= C_i^{512}\times C_i^{64}\times C_i^{32}\times C_i^{8}\times C_i^{2}\times C_i^{1}\times$
   e le potenze si calcolano facilmente secondo la seguente tabella, relativa a $C_i = 328$:

k	**C_1^k mod 1003**
$2^0=1$	$328^1 \bmod 1003 = \mathbf{328}$
$2^1=2$	$328^2 \bmod 1003 = \mathbf{263}$
$2^2=3$	$263^2 \bmod 1003 = 965$
$2^3=8$	$965^2 \bmod 1003 = \mathbf{441}$
$2^4=16$	$441^2 \bmod 1003 = 902$
$2^5=32$	$902^2 \bmod 1003 = \mathbf{171}$
$2^6=64$	$171^2 \bmod 1003 = \mathbf{154}$
$2^7=128$	$154^2 \bmod 1003 = 647$
$2^8=256$	$647^2 \bmod 1003 = 358$
$2^9=512$	$358^2 \bmod 1003 = \mathbf{783}$

   dove abbiamo messo in neretto i fattori che dobbiamo moltiplicare. Dalla tabella si vede facilmente che

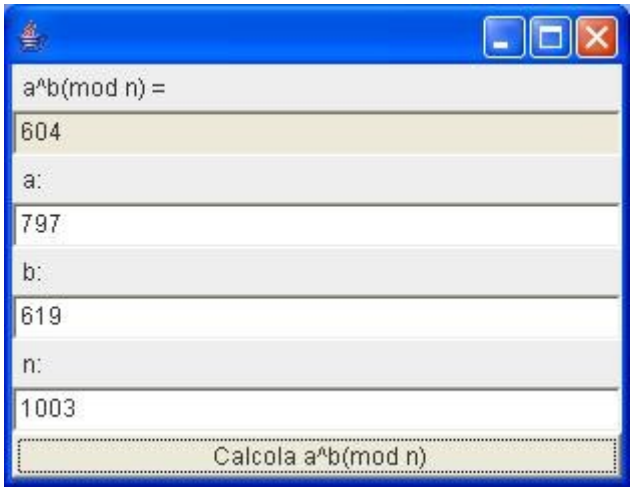
$$328^{619} = 328^{512} \times 328^{64} \times 328^{32} \times 328^{8} \times 328^{2} \times 328^{1}$$
$$\equiv 783 \times 154 \times 171 \times 441 \times 263 \times 783 \times 328 = 784416598601328 \equiv 11 \pmod{1003}$$

Notiamo che, elevando $C_1 = 328$ all'esponente $d_B = 619$, abbiamo ottenuto il numero $11 = P_1$, cioè il numero corrispondente al messaggio originario.

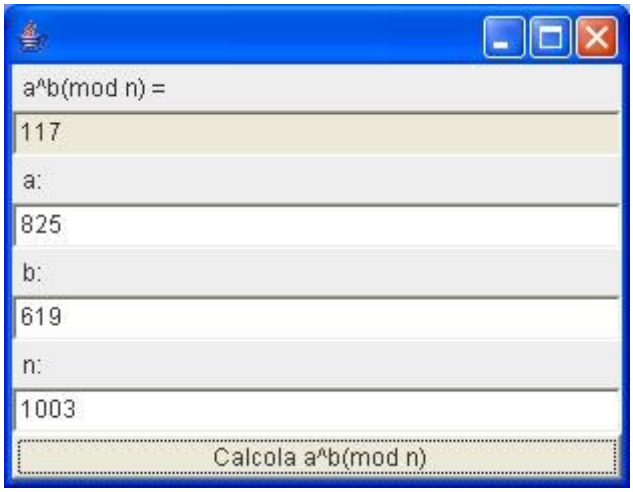


Procediamo allo stesso modo, con gli altri tre segmenti di messaggio C_2 , C_3 e C_4 , ovvero:

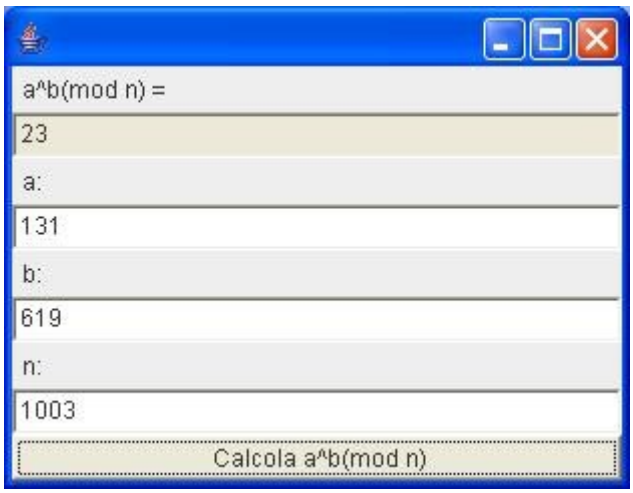
- $C_2^{\,dB} \bmod n_B = 797^{619} \bmod 1003 = 604$



- $C_3^{\,dB} \bmod n_B = 825^{619} \bmod 1003 = 117$



- $C_4^{\,dB} \bmod n_B = 131^{619} \bmod 1003 = 23$



2. i risultati ottenuti vanno gestiti a quattro cifre, ossia

   0011 0604 0117 0023

   e corrispondono a quattro segmenti di messaggio originali, che erano dei digrafi.

3. Dobbiamo quindi dividere ciascuno in due parti e ciascuna delle due parti rappresenta una lettera. L'utente *B* pertanto, utilizza il criterio precedentemente illustrato nel paragrafo introduttivo all'impostazione matematica di un crittosistema. Quindi la sequenza di numeri corrispondenti al nostro messaggio sarà :

00	11	06	04	01	17	00	23
a	*l*	*g*	*e*	*b*	*r*	*a*	*x*

Perché ha funzionato? Cioè perché elevando C_i all'esponente d_B si riottiene P_i tale che $C_i = P_i^{eB}$?

Perché ha funzionato?

Come già è stato osservato in precedenza, congruenza $e_B\, d_B \equiv 1 \ (mod\ \varphi(n_B))$ ammette un'unica soluzione , poiché $MCD(e_B, p_B-1)=MCD(e_B, q_B-1)=MCD(e_B, (p_B-1)(q_B-1))=MCD(e_B, \varphi(n_B))=1$.

Osserviamo inoltre che l'utente *B* è l'unico che può risolvere tale congruenza poiché è l'unico che conosce la funzione di Eulero $\varphi(n_B)=(p_B-1)(q_B-1)$, conoscendo i fattori primi p_B e q_B di n_B. Notiamo infatti che, essendo p_B e q_B numeri primi grandi, la fattorizzazione di n_B richiede, normalmente, un tempo di calcolo enorme. Quindi, di fatto, *B* è come se fosse l'unico a conoscere la fattorizzazione di n_B.

Verifichiamo ora che d_B è proprio la chiave segreta che consente all'utente *B* di decifrare il messaggio. Ponendo infatti $P = P_i$ e $C = C_i$, si ha

$$P \equiv C^{d_B} (mod\ n_B)$$

perché

$$C^{d_B} \equiv \left(P^{e_B}\right)^{d_B} = P^{e_B d_B} (mod\ n_B)$$

inoltre, poiché l'utente *A*, ha scelto $P < n_B$, non c0è ambiguità nella determinazione del numero congruo a C^{dB} modulo n_B : si tratta dell'unico numero siffatto compreso tra *0 e* n_B*-1*.

Una miglioria dell'algoritmo

Abbiamo visto che, per inviare il messaggio *algebra* all'utente *B*, l'utente *A* ha suddiviso il testo in

digrafi. Per fare ciò ha dovuto procedere per tentativi, verificando che i numeri corrispondenti ai singoli digrafi fossero tutti minori di n_B e primi con esso.

C'è però un metodo migliore per scegliere in che modo suddividere il messaggio: conviene infatti non dividere il messaggio originario, ma bensì il suo corrispettivo numerico ottenuto associando un numero a due cifre ad ogni lettera. In questa maniera, come vedremo, c'è un modo naturale di suddividerlo.

Si procede così: una volta che si trasforma il messaggio testuale nell'equivalente numerico a due cifre, consideriamo il numero costituito dalla successione di tutte le cifre, che chiameremo *messaggio numerico*. Suddividiamolo poi in blocchi di k cifre tale che k corrisponde al numero di cifre di n_B diminuito di un'unità.

In tale modo, *senza dover esaminare il messaggio*, il singolo messaggio numerico unitario è automaticamente minore di n_B. Non solo, ma questo modo di procedere è noto a chiunque, perché tutti conoscono n_B e sanno che il mittente del messaggio dividerà il messaggio numerico in blocchi siffatti. A tal proposito, consideriamo l'esempio a seguire.

Supponiamo che l'utente *A*, debba inviare all'utente *B* il messaggio *'vieni qui'*. Allora procediamo nella seguente maniera:

1. Per prima cosa trasformare tutte le lettere del messaggio in un intero in base all'equivalenza numerica a due cifre, secondo il criterio precedentemente illustrato nel paragrafo introduttivo all'impostazione matematica di un crittosistema. Quindi la sequenza di numeri corrispondenti al nostro messaggio sarà :

v	*i*	*e*	*n*	*i*	*q*	*u*	*i*
21	*08*	*04*	*13*	*08*	*16*	*20*	*08*

   Che scriveremo come

   2108041308162008

   ciò non crea ambiguità perché sappiamo che ogni numero associato ad una lettera del messaggio originario è costituita da due cifre. Questo è il *messaggio numerico*.

2. Successivamente l'utente *A* consulterà l'elenco pubblico delle chiavi RSA, dal quale ricaverà la coppia (n_B , e_B) associata ovviamente all'utente *B*. Per il momento serve solo $n_B = 1003$, il quale è composto da *4* cifre, quindi l'utente *A* spezza il messaggio numerico in gruppi di *4-1=3* cifre, ossia *trigrafi*, come segue:

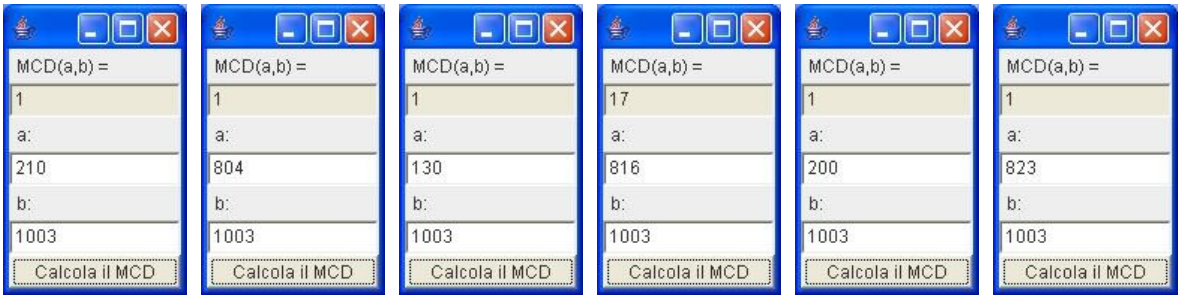
   210 804 130 816 200 823

   Si noti come l'utente *A* abbia aggiunto *23* in coda al messaggio numerico, quale corrispettivo della lettera *x*, in modo tale che tutti i messaggi numerici unitari siano costituiti da tre cifre. In questo modo ha spezzato il messaggio numerico in messaggi numerici unitari che sono dei trigrafi e ogni messaggio unitario è sicuramente minore di $n_B = 1003$.

   Si osservi che questa suddivisione non è una suddivisione del messaggio originario *'vieni qui'* perché i blocchi numerici unitari di *3* cifre non corrispondono a nessun gruppo di lettere. Vedremo che questo non creerà nessun problema.

3. A questo punto si deve controllare che ogni P_i sia relativamente primo con *1003*.

210 *804* *130* *816* *200* *823*



Si verifica agevolmente che l'unica eccezione è quella di *816*. tuttavia non è il caso di preoccuparsi!

In definitiva i singoli messaggi numerici unitari sono :

- $P_1 = 210$
- $P_2 = 804$
- $P_3 = 130$
- $P_4 = 816$
- $P_5 = 200$
- $P_6 = 823$

Si noti che le operazioni fin qui effettuate non corrispondono ad alcun tipo di cifratura, perché la trasformazione di un messaggio in una successione di numeri è standard così come la ripartizione dello stesso in gruppi di tre cifre, che dipende dal valore di n_B che tutti conoscono.

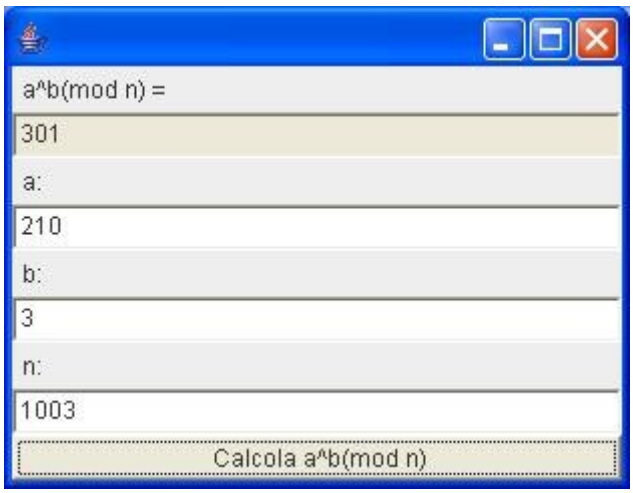
4. Ora ha inizio l'operazione di cifratura vera e propria per l'invio del messaggio all'utente *B*, in modo tale che quest'ultimi lo possa *decifrare e sia l'unica a poterlo fare in un tempo ragionevole*[10]. Per spedire il messaggio da spedire all'utente *B*, l'utente *A* eleva ogni P_i alla potenza e_B , essendo e_B il secondo elemento della coppia associata all'utente *B*.
La funzione di cifratura risulta essere quindi

$$C_B : P \rightarrow C \qquad P \rightarrow C = P^{e_B} \bmod n_B$$

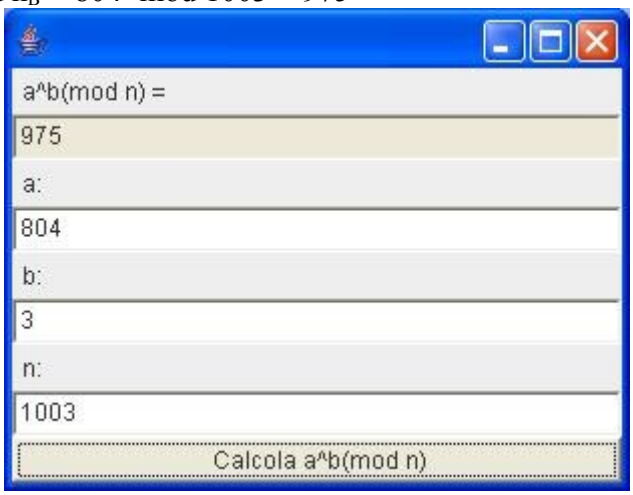
il messaggio cifrato che l'utente *A* deve mandare sarà dunque rappresentato dai seguenti numeri:

- $C_1 = P_1^{\,eB} \bmod n_B = 210^3 \bmod 1003 = 301$

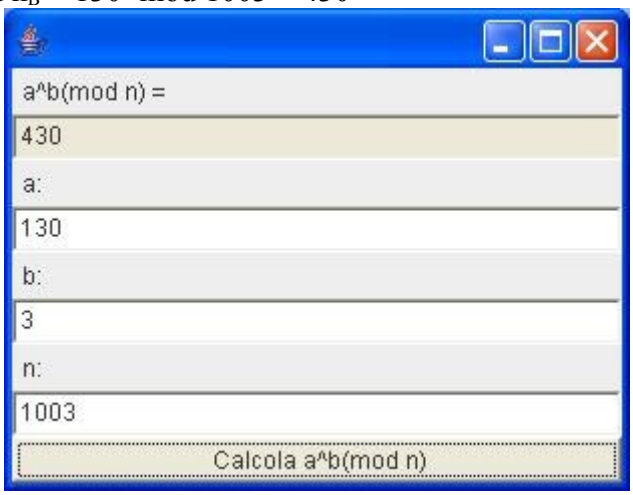
10 Tempo polinomiale



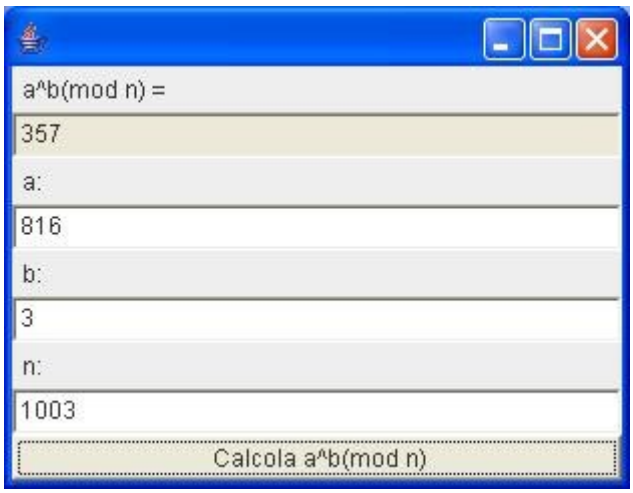
- $C_2 = P_2^{eB} \bmod n_B = 804^3 \bmod 1003 = 975$



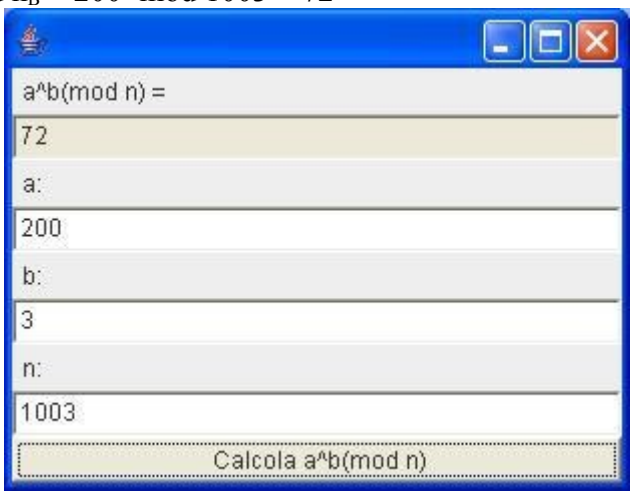
- $C_3 = P_3^{eB} \bmod n_B = 130^3 \bmod 1003 = 430$



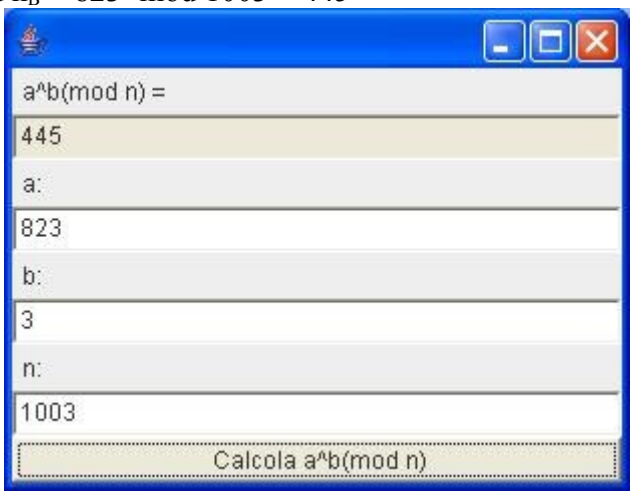
- $C_4 = P_4^{eB} \bmod n_B = 816^3 \bmod 1003 = 357$



- $C_5 = P_5^{eB} \bmod n_B = 200^3 \bmod 1003 = 72$



- $C_6 = P_6^{eB} \bmod n_B = 823^3 \bmod 1003 = 445$



Si noti, anche in questo caso, come tali valori siano stati calcolati direttamente con la classe *TestModPow.java*, illustrata nel capitolo precedente.

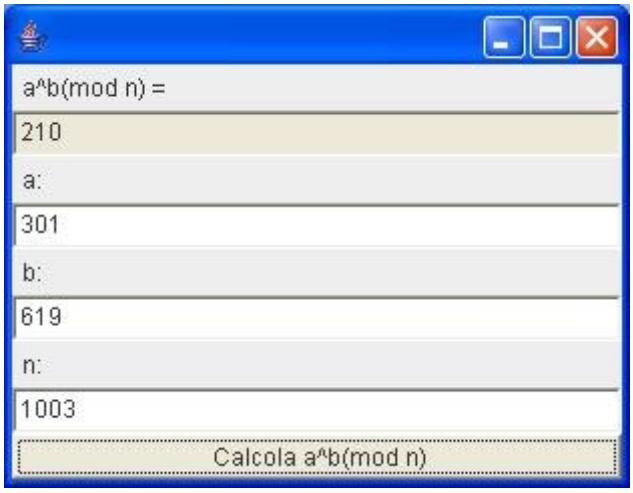
L'utente *B* riceve quindi il messaggio:

- $C_1 = 301$
- $C_2 = 975$
- $C_3 = 430$

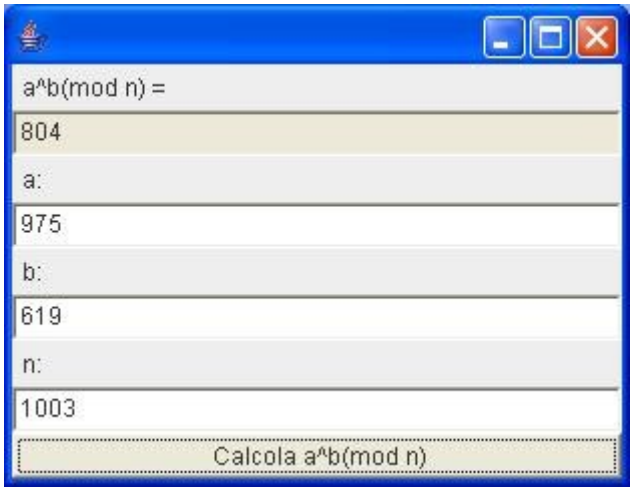
- $C_4 = 357$
- $C_5 = 72$
- $C_6 = 445$

Costituito dai numeri unitari C_i , $i = 1, \ldots, 6$. A questo punto l'utente A ha terminato il suo compito: ha inviato all'utente B il messaggio cifrato. Ora sarà compito dell'utente B decifrarlo.

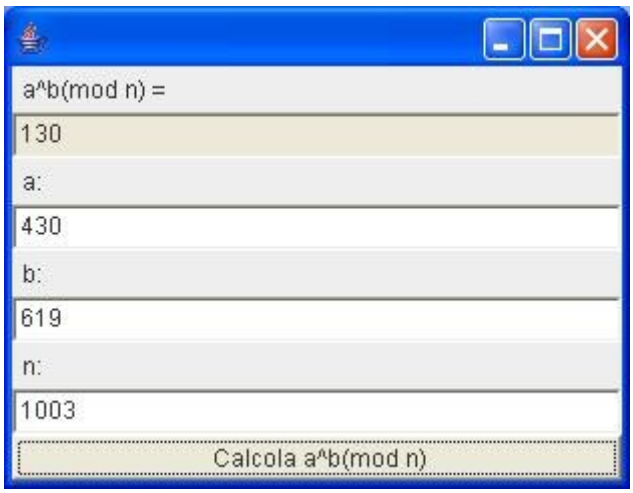
5. Per decifrare, l'utente B, eleva ogni C_i alla sua chiave *segreta* $d_B = 619$, cioè calcola:
   - $C_1^{dB} \bmod n_B = 301^{619} \bmod 1003 = 210$



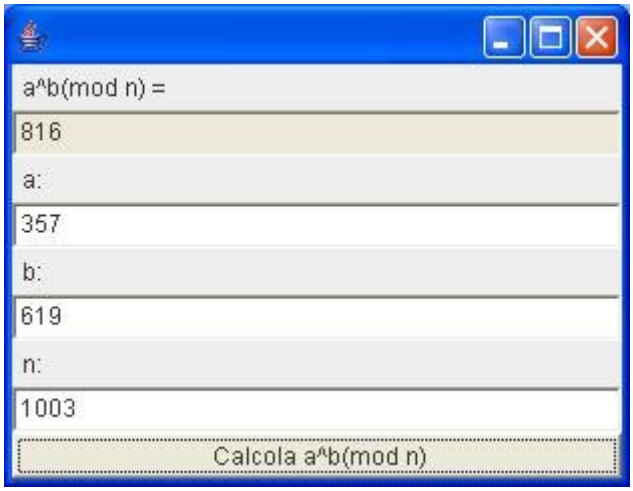
   - $C_2^{dB} \bmod n_B = 975^{619} \bmod 1003 = 804$



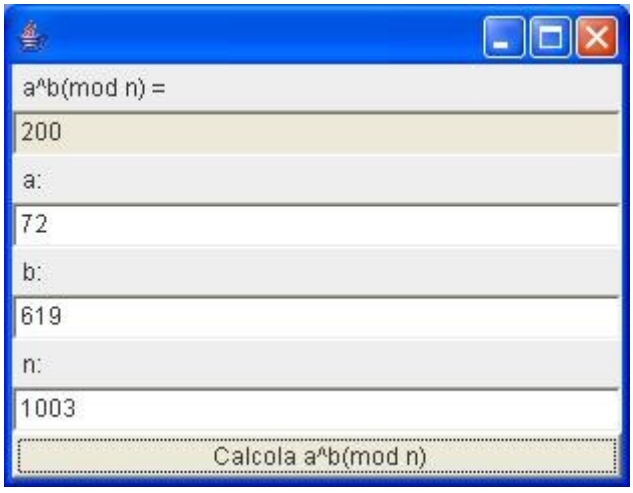
   - $C_3^{dB} \bmod n_B = 430^{619} \bmod 1003 = 130$



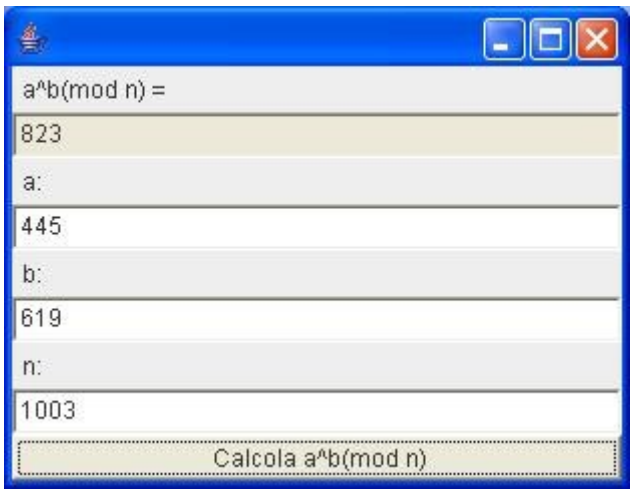
- $C_4^{\,dB} \bmod n_B = 357^{619} \bmod 1003 = 816$



- $C_5^{\,dB} \bmod n_B = 72^{619} \bmod 1003 = 200$



- $C_6^{\,dB} \bmod n_B = 445^{619} \bmod 1003 = 823$

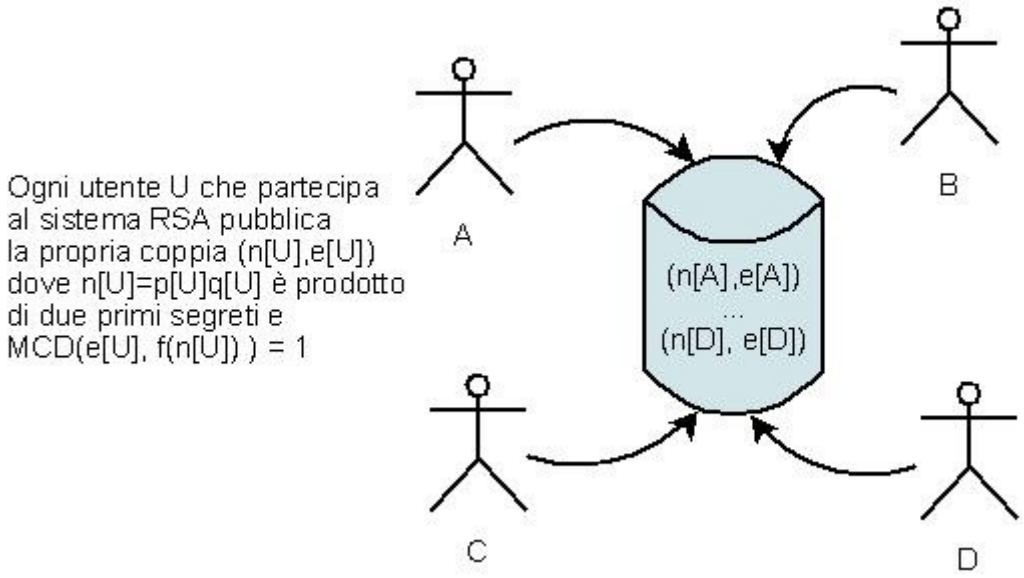


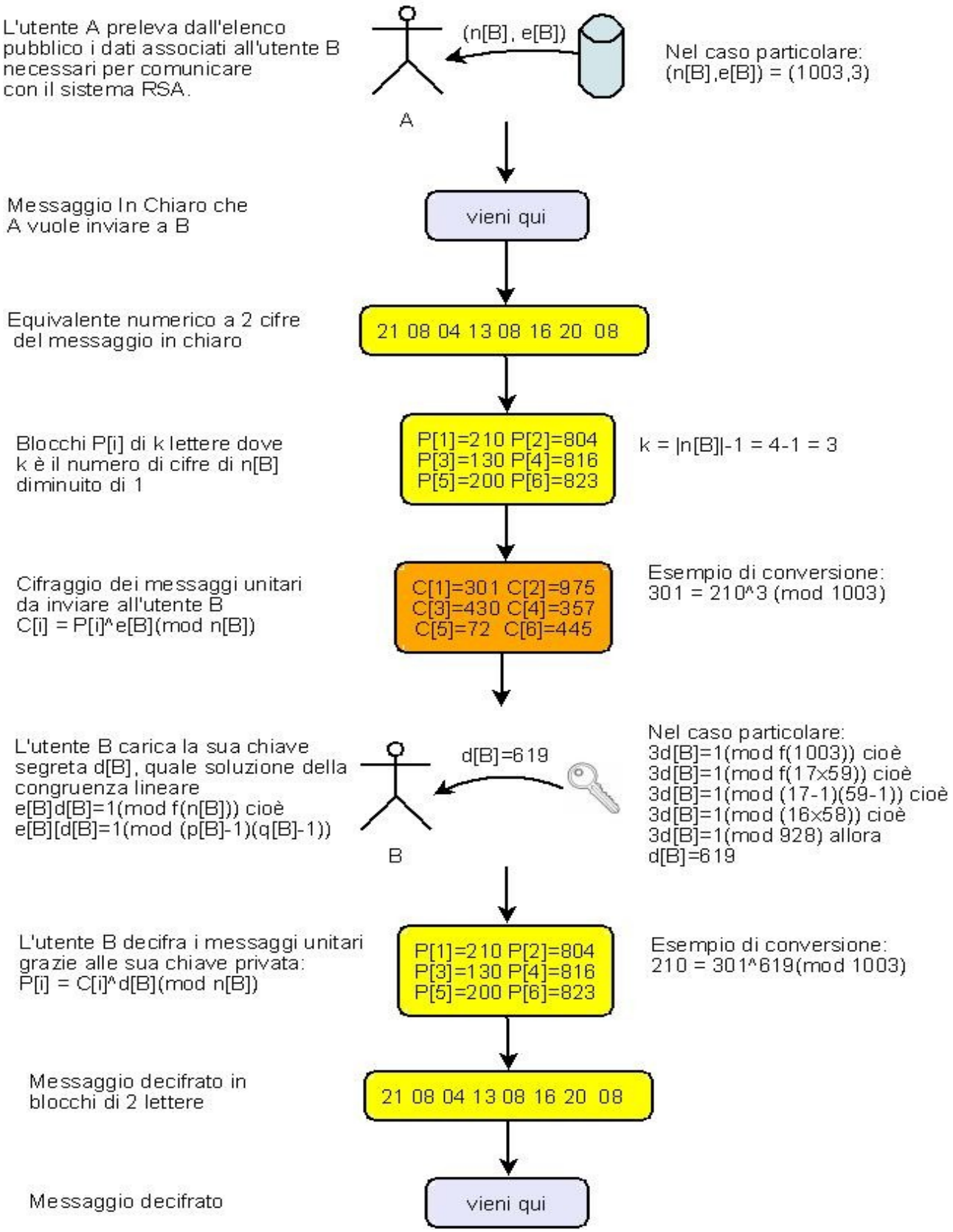
questi gruppi di tre cifre, raggruppati a due a due, danno:

21	*08*	*04*	*13*	*08*	*16*	*20*	*08*	*23*
v	*i*	*e*	*n*	*i*	*q*	*u*	*i*	*x*

e quindi l'utente *B* riesce a leggere il messaggio *vieni quix* che capisce significare *vieni qui*.

L'esempio proposto è rappresentato nel seguente schema logico :





Autenticazione delle firme con il sistema RSA

Uno dei problemi significativi affrontati nei paragrafi precedenti di questo capitolo, consiste nel garantire l'autenticità e quindi la non ripudiabilità di un messaggio da parte di un mittente. Come si

è visto uno dei modelli informatici attuati per affrontare tale problematica coincide con l'applicazione della firma digitale. In questo paragrafo, analizzeremo il modello matematico di tale architettura compatibilmente con le esigenze strutturali del sistema RSA.

Se l'utente *B* riceve un messaggio da una persona che si firma come utente *A*, come può essere certo che sia stato proprio quest'ultimi ad inviargli la missiva? La garanzia la si ottiene procedendo come segue:

1. L'utente *A* scrive il suo messaggio P_1 , in fondo al quale comparirà la sua firma *F*. per autenticare la firma, l'utente *A* aggiunge in fondo al messaggio P_1 il messaggio

   $$P_2 = F^{a_A} (mod\ n_A)$$

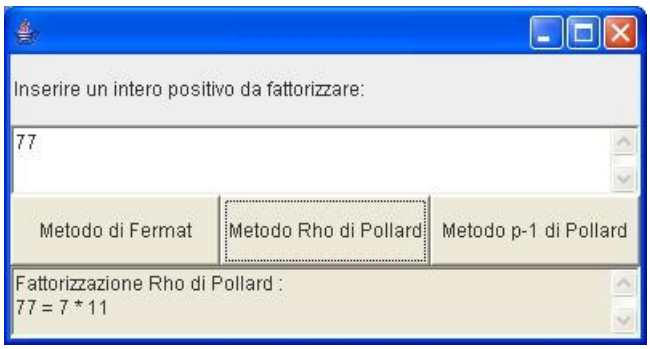
   dove d_A è la sua chiave segreta, ossia quella che lui conosce, perché conosce la fattorizzazione della propria chiave *pubblica* n_A.
2. Realizzata la firma, manda all'utente *B* il messaggio *P* composto da due messaggi P_1 e P_2 al solito modo, ossia elevando P_1 e P_2 alla potenza e_B e riducendoli modulo n_B.
3. Nel ricevere il messaggio, l'utente *B* lo legge utilizzando la propria chiave segreta d_B .
4. Dalla decifrazione del messaggio P_1 , capisce che il messaggio gli è stato inviato dall'utente *A*, poiché il messaggio è firmato con la firma *F* dell'utente *A*. Il tutto consiste nell'essere sicuri che sia stato proprio l'utente *A* e non un altro, ad avere apposto quella firma!
5. Per assicurarsi della genuinità del mittente, ci viene in aiuto l'altra parte del messaggio *P*, ovvero il blocco P_2. Infatti in questi sono presenti dei caratteri indecifrabili, che però contengono la prova dell'autenticità della firma. Ora, l'utente *B* per decifrare P_2 non deve utilizzare la propria chiave segreta d_B , la quale non gli sarebbe di alcuna utilità, in quanto il messaggio originario *F* è stato crittografato elevandolo non alla potenza e_B ma bensì a quella d_A. L'utente *B* utilizza invece la chiave pubblica e_A dell'utente *A*. In tal modo *viene fuori la firma F* dell'utente *A*, poiché:

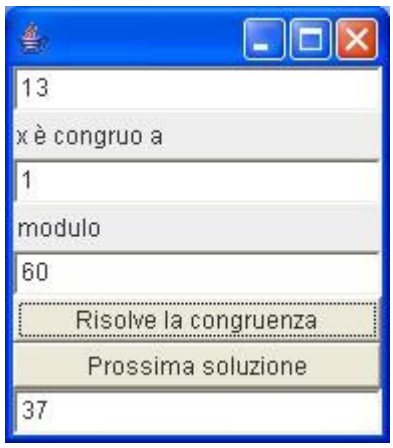
   $$\left(P_2\right)^{e_A} \equiv \left(F^{d_A}\right)^{e_A} = F^{d_A e_A} \equiv F\,(mod\ n_A)$$

   Tale firma non può essere che quella autentica, perché solamente l'utente *A* è a conoscenza della propria chiave segreta. Nel caso in cui non fosse comparsa la firma *F* dell'utente *A*, il messaggio sarebbe stato un falso, ovvero violato nella sua autenticità e riservatezza.

In sostanza, per l'autenticazione di una firma è il *mittente* che utilizza la *propria chiave segreta*, anziché il *destinatario*.

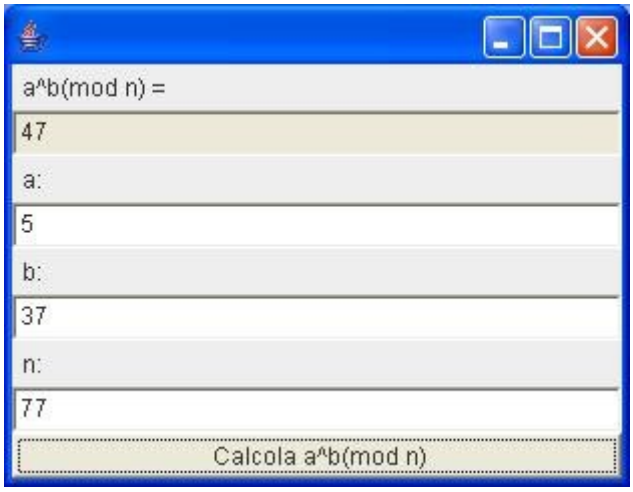
Riprendiamo l'esempio esposto in precedenza, in cui l'utente *A* manda il messaggio '*vieni qui*' all'utente *B*. Abbiamo detto che l'utente *A* per firmare il messaggio deve utilizzare la propria chiave privata d_A, la quale la ricava dalla risoluzione della congruenza lineare $e_A d_A \equiv 1\ (mod\ \varphi(n_A))$ tale che $\varphi(n_A) = (p_A - 1)(q_A - 1)$ ovvero $\varphi(77) = (11-1)(7 - 1)=60$, quindi sostituendo i valori la congruenza lineare da risolvere diventa $13\, d_A \equiv 1\ (mod\ 60)$ la cui soluzione è $d_A = 37$.





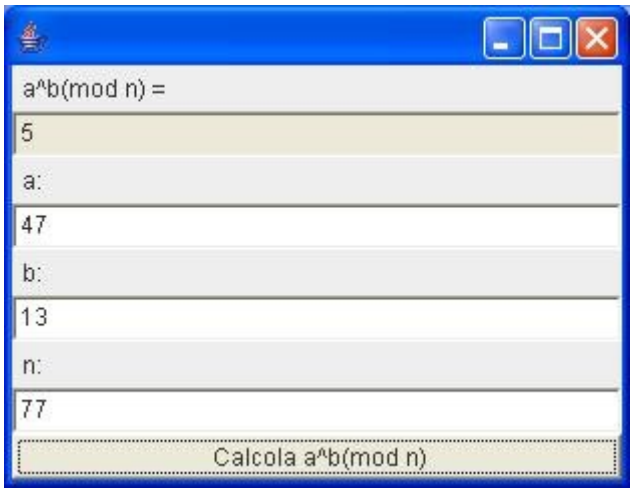
Ipotizziamo che nel messaggio l'utente A abbia apposto la firma $F=5$, elevando quest'ultimi alla potenza $d_A(mod\ n_A)$, otteniamo:

$$P_2 = F^{dA}(mod\ n_A) = 5^{37}(mod\ 77) = 47$$



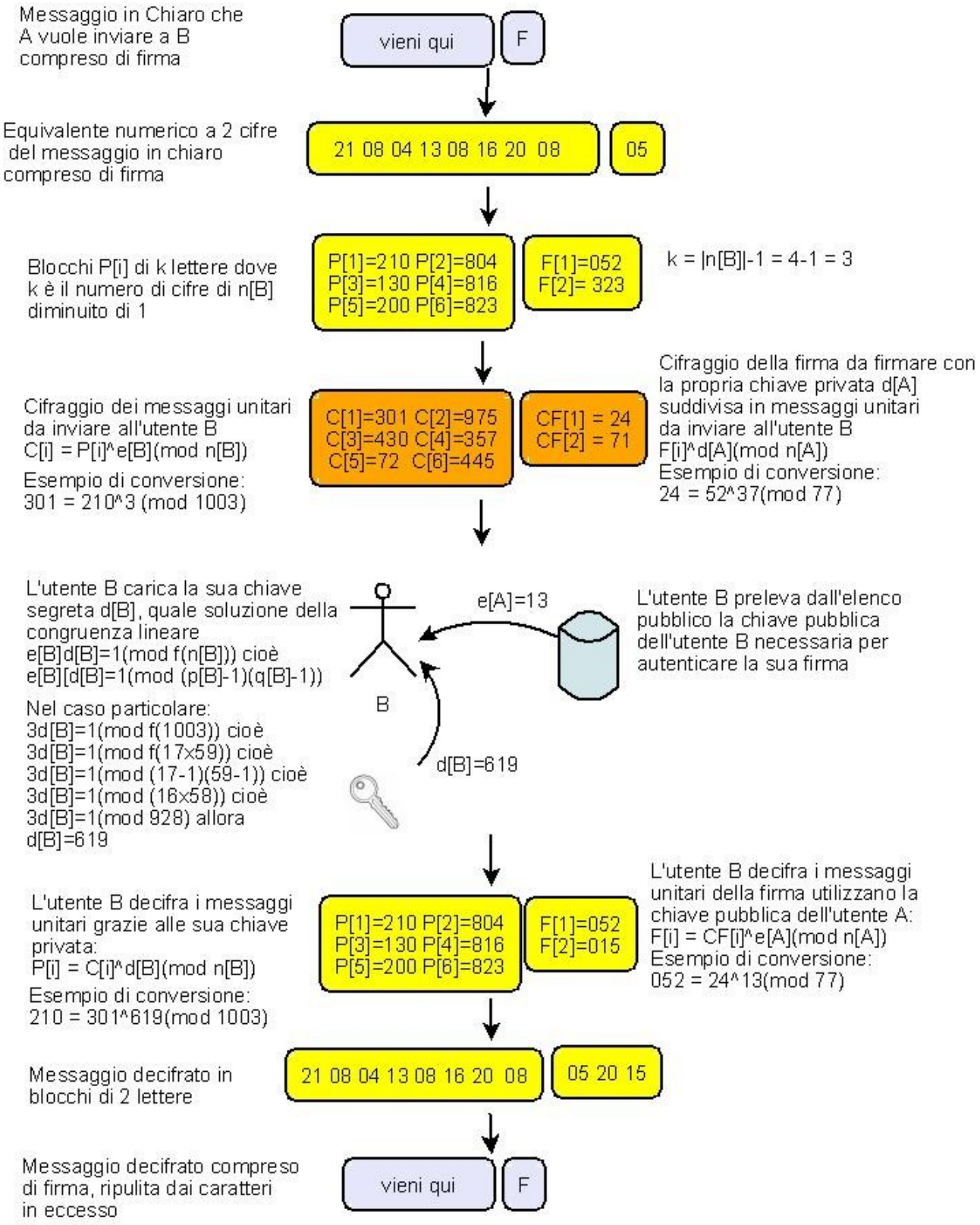
L'utente B verifica l'autenticità della firma elevando 47 alla potenza $e_A = 13(mod\ n_A)$:

$$P_2{}^{eA}(mod\ n_A) = 47^{13}(mod\ 77) = 5$$



cioè riottiene la firma dell'utente A. In tal modo l'utente B è sicuro che il messaggio proviene dall'utente A.

La metodologia RSA compresa di sistema di firma, viene riassunta nel seguente schema logico :



L'implementazione

La metodologia RSA dimostrata in precedenza è stata implementata nella classe *Chipers.java*, con i rispettivi metodi *RSAEncipher(msg, e, n)* ed *RSADecipher(msg, e, n)* :

```
/**
```

```
 * Cripta secondo il sistema RSA
 * @param msg di tipo byte[], rappresenta il messaggio in chiaro espresso in
byte
 * @param e di tipo BigInteger, rappresenta la chiave pubblica
 * @param n di tipo BigInteger, rappresenta il modulo
 * @return un array di byte
 * */
public static byte[] RSAEncipher(byte[] msg,BigInteger e,BigInteger n) {

      //Determina la grandezza in blocchi del testo in chiaro
      int blockSize=(n.bitLength()-1)/8;
      byte[][] ba=block(pad(msg,blockSize),blockSize);
      //Avvia il processo di crittazione
      for (int i=0;i<ba.length;i++)
            ba[i]=getBytes(new BigInteger(1,ba[i]).modPow(e,n));

      //il blocco cifrato è di una dimensione maggiore di byte rispetto al testo
in chiaro.
      return unBlock(ba,blockSize+1);
}//RSAEncipher

/**
 * Decripta secondo il sistema RSA
 * @param msg di tipo byte[], rappresenta il messaggio criptato espresso in byte
 * @param d di tipo BigInteger, rappresenta la chiave privata
 * @param n di tipo BigInteger, rappresenta il modulo
 * @return un array di byte
 * */
public static byte[] RSADecipher(byte[] msg,BigInteger d,BigInteger n) {

      //Calcola la grandezza del blocchi criptati
      int blockSize=(n.bitLength()-1)/8+1;
      byte[][] ba=block(msg,blockSize);
      //Avvia la decrittazione
      for (int i=0;i<ba.length;i++)
            ba[i]=getBytes(new BigInteger(1,ba[i]).modPow(d,n));

      //spacchetta il testo
      return unPad(unBlock(ba,blockSize-1),blockSize-1);
}//RSADecipher
```

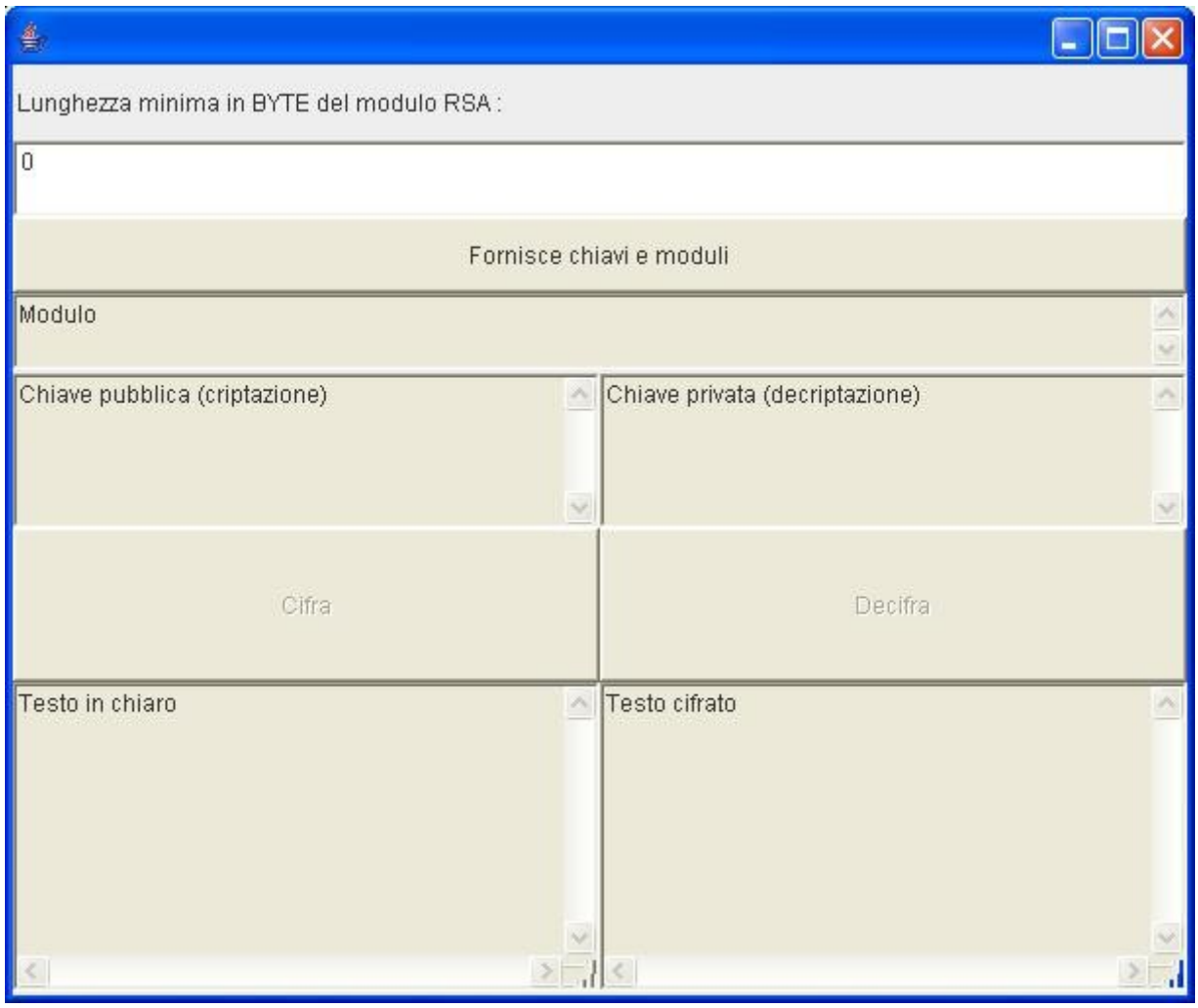
i quali sono stati testati con la classe *TestRSAChiper.java*:

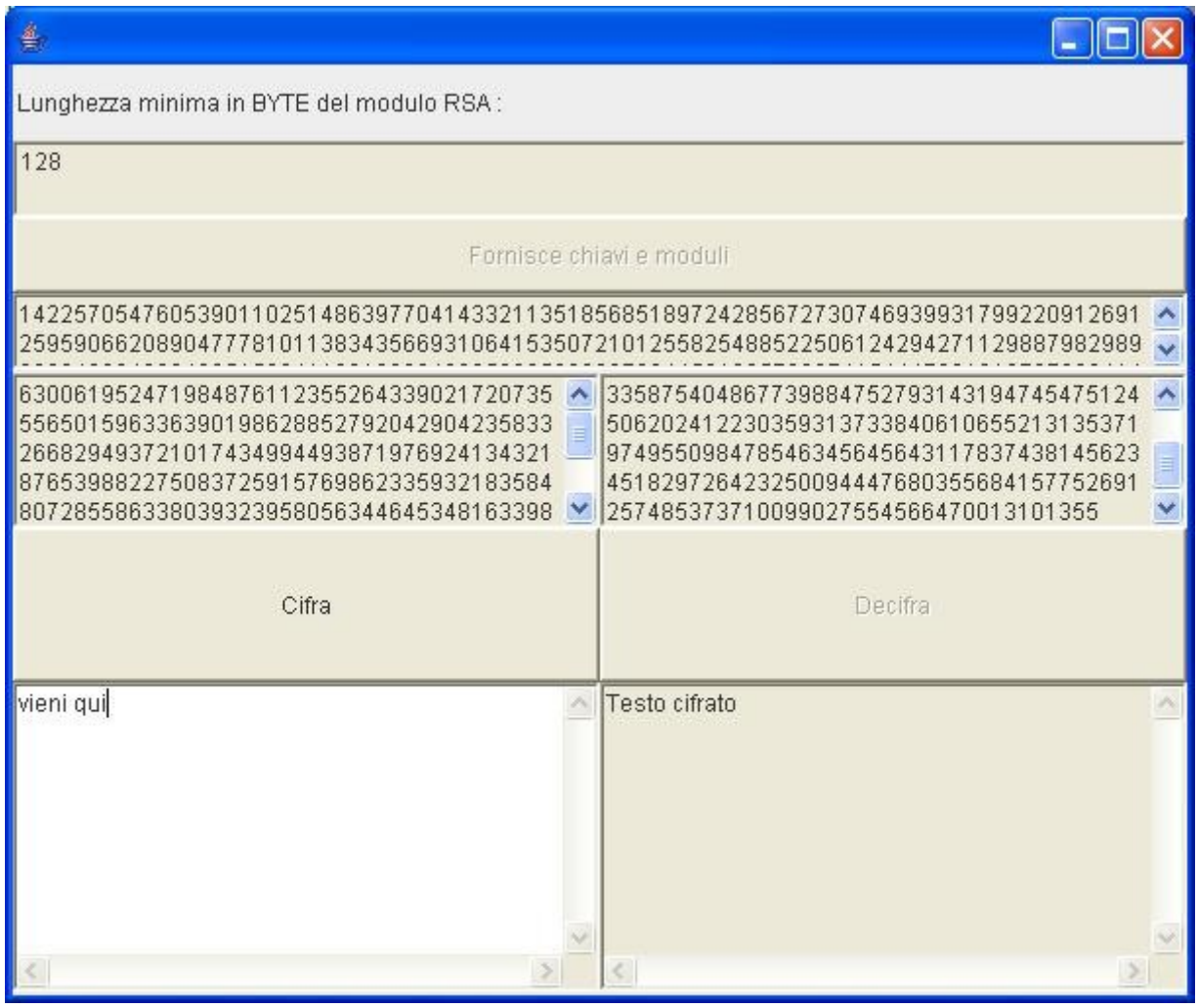
```
package cifrari;

import java.math.*;
import java.awt.*;
import java.awt.event.*;
import java.security.*;
import javax.swing.*;

import algebra.BigIntegerMath;
import algebra.PrimeGenerator;

public class TestRSACipher extends JFrame implements ActionListener {

   BigInteger dexponent=null;
   BigInteger exponent=null;
   int modByteLength=0;
   BigInteger p=null,q=null,modulus=null;
   byte[] msgArray=null;
```

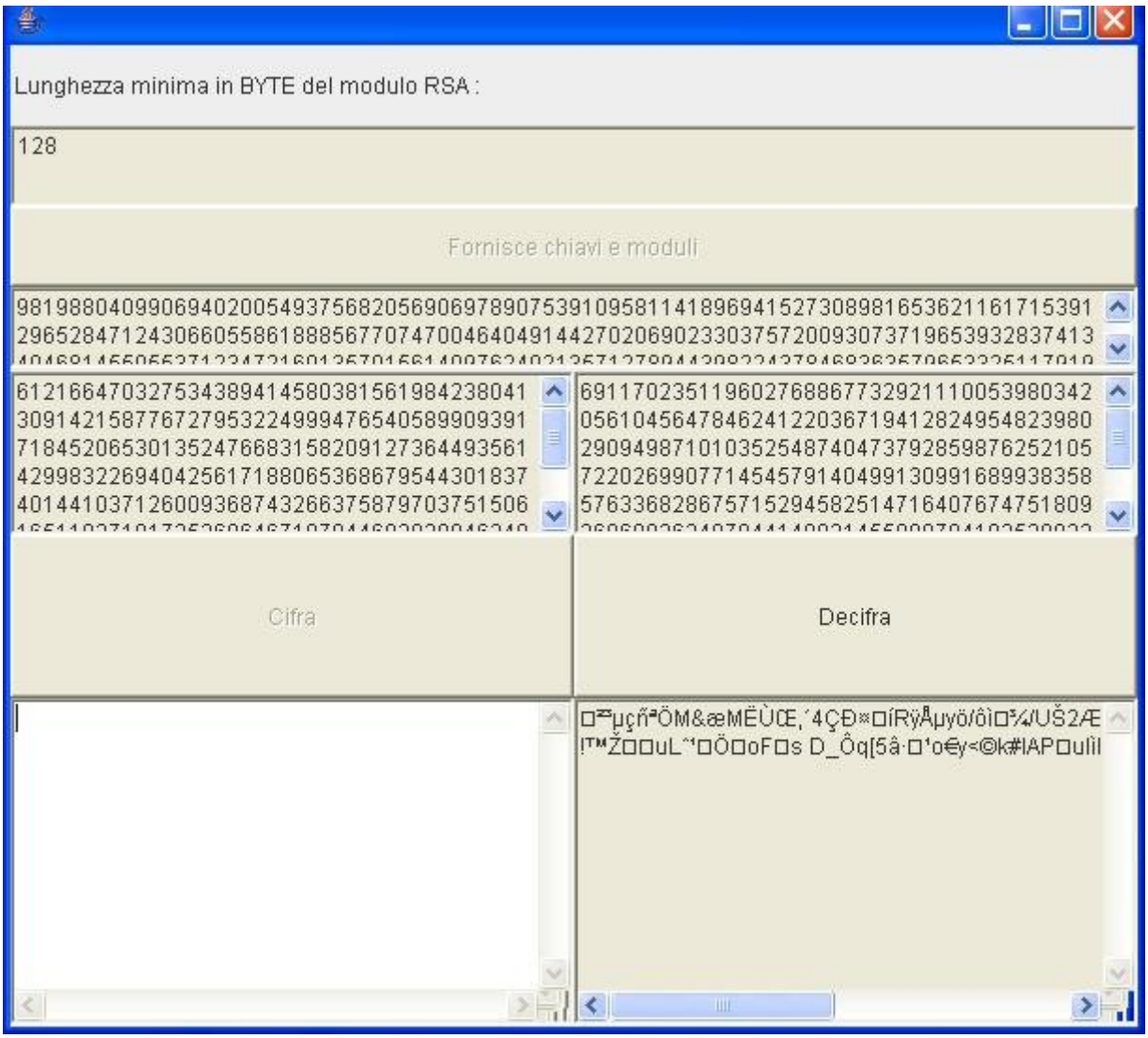
```
   byte[] encmsgArray=null;

   Label byteLengthLabel=new Label("Lunghezza minima in BYTE del modulo RSA :");
   TextField byteLengthField=new TextField("0");
   Button getKeysButton=new Button("Fornisce chiavi e moduli");
   TextArea modArea=new TextArea("Modulo",
10,40,TextArea.SCROLLBARS_VERTICAL_ONLY);
   TextArea eArea=new TextArea("Chiave pubblica (criptazione)",
10,40,TextArea.SCROLLBARS_VERTICAL_ONLY);
   TextArea dArea=new TextArea("Chiave privata (decriptazione)",
10,40,TextArea.SCROLLBARS_VERTICAL_ONLY);
   TextArea plaintextArea=new TextArea("Testo in chiaro",
10,40,TextArea.SCROLLBARS_BOTH);
   TextArea ciphertextArea=new TextArea("Testo cifrato",
10,40,TextArea.SCROLLBARS_BOTH);
   Button encButton=new Button("Cifra");
   Button decButton=new Button("Decifra");
   Panel topPanel=new Panel();
   Panel centerPanel=new Panel();
   Panel bottomPanel=new Panel();

   SecureRandom sr=null;

   public TestRSACipher() {

         Panel p1=new Panel();

         p1.setLayout(new GridLayout(3,1));

         topPanel.setLayout(new GridLayout(4,1));
         topPanel.add(byteLengthLabel);
         topPanel.add(byteLengthField);
         topPanel.add(getKeysButton);
         getKeysButton.addActionListener(this);
         topPanel.add(modArea);
         modArea.setEditable(false);

         p1.add(topPanel);

         centerPanel.setLayout(new GridLayout(2,2));
         centerPanel.add(eArea);
         eArea.setEditable(false);
         centerPanel.add(dArea);
         dArea.setEditable(false);
         centerPanel.add(encButton);
         encButton.setEnabled(false);
         encButton.addActionListener(this);
         centerPanel.add(decButton);
         decButton.setEnabled(false);
         decButton.addActionListener(this);

         p1.add(centerPanel);

         bottomPanel.setLayout(new GridLayout(1,2));
         bottomPanel.add(plaintextArea);
         plaintextArea.setEditable(false);
         bottomPanel.add(ciphertextArea);
         ciphertextArea.setEditable(false);

         p1.add(bottomPanel);
```

```
        add(p1);
        pack();
        show();
  }

    public void actionPerformed(ActionEvent e) {
    Object source=e.getSource();
       if (source==getKeysButton) {
          try {
                modByteLength=Integer.parseInt(byteLengthField.getText());
                sr=new SecureRandom();
                //Utilizza la classe PrimeGenrator del package algebra per
                //ottenere un numero primo sicuro
                PrimeGenerator pg=new PrimeGenerator(modByteLength*4+1,10,sr);
                p=pg.getStrongPrime();
                q=pg.getStrongPrime();
                modulus=p.multiply(q);
                modArea.setText(modulus.toString());
                BigInteger pMinusOne=p.subtract(BigIntegerMath.ONE);
                BigInteger qMinusOne=q.subtract(BigIntegerMath.ONE);
                do {
                   exponent=new BigInteger(modulus.bitLength()-1,10,sr);
                } while (!
exponent.gcd(pMinusOne.multiply(qMinusOne)).equals(BigIntegerMath.ONE));
                dexponent=exponent.modInverse(pMinusOne.multiply(qMinusOne));
                eArea.setText(exponent.toString());
                dArea.setText(dexponent.toString());
                getKeysButton.setEnabled(false);
                byteLengthField.setEditable(false);
                plaintextArea.setEditable(true);
                encButton.setEnabled(true);
          } catch (Exception nfe) {
             byteLengthField.setText(nfe.toString());
     }
       } else if (source==encButton) {
          msgArray=plaintextArea.getText().getBytes();
          //Sempre la stessa chiave
          encmsgArray=Ciphers.RSAEncipher(msgArray,exponent,modulus);
          ciphertextArea.setText(new String(encmsgArray));
          plaintextArea.setText("");
          encButton.setEnabled(false);
          decButton.setEnabled(true);
       } else if (source==decButton) {
        //con chiave fissa
        msgArray=Ciphers.RSADecipher(encmsgArray,dexponent,modulus);
          plaintextArea.setText(new String(msgArray));
          ciphertextArea.setText("");
          decButton.setEnabled(false);
          encButton.setEnabled(true);
    }
  }

    public static void main(String[] arg){
            new TestRSACipher();
       }//main

}
```

il cui output realizzato mediante l'uso del codice ASCII, risulta essere il seguente:


Lunghezza minima in BYTE del modulo RSA :
0
Fornisce chiavi e moduli
Modulo
Chiave pubblica (criptazione)
Chiave privata (decriptazione)
Cifra
Decifra
Testo in chiaro
Testo cifrato


Lunghezza minima in BYTE del modulo RSA :
128
Fornisce chiavi e moduli
1422570547605390110251486397704143321135185685189724285672730746939931799220912691
2595906620890477781011383435669310641535072101255825488522506124294271129887982989
630061952471984876112355264339021720735
556501596336390198628852792042904235833
266829493721017434994493871976924134321
876539882275083725915769862335932183584
807285586338039323958056344645348163398
335875404867739884752793143194745475124
506202412230359313733840610655213135371
974955098478546345645643117837438145623
451829726423250094447680356841577526 91
257485373710099027554566470013101355
Cifra
Decifra
vieni qui
Testo cifrato



I principali algoritmi di crittografia a chiave pubblica

Oltre all'ampiamente discusso sistema RSA, esistono anche altri algoritmi a chiave pubblica, di cui alcuni semplici varianti dello stesso RSA:

- **DSS (Digital Signature Standard)** : proposto dal NIST (National Institute of Standard and Technology) nel 1991. È un algoritmo basato sul metodo dei logaritmi discreti ed è valido solo per l'autenticazione e l'integrità. È uno standard obbligatorio per i sistemi del governo federale statunitense.
- **Diffie-Helmann** : è un sistema crittografico che permette di scambiarsi una chiave segreta tramite un canale non sicuro senza la conoscenza preventiva di alcuna informazione segreta. Fu sviluppato nel 1976 da Diffie e da Helmann ed è il primo algoritmo a chiave pubblico della storia. L'efficacia dell'algoritmo dipende dalla difficoltà di calcolare i logaritmi discreti. Questo protocollo è vulnerabile all'attacco noto come *man in the middle*, infatti un intruso può posizionarsi tra un mittente ed un destinatario e scambiare le loro chiavi pubbliche e private con altre opportunamente modificate.
- **El Gamal** : è un algoritmo di cifratura anch'esso basato sui logaritmi discreti. Estende il metodo di Diffie-Helmann verso la cifratura e la firma digitale. El Gamal e RSA hanno una sicurezza simile per lunghezze di chiavi equivalenti. Il primo è però più lento del secondo.

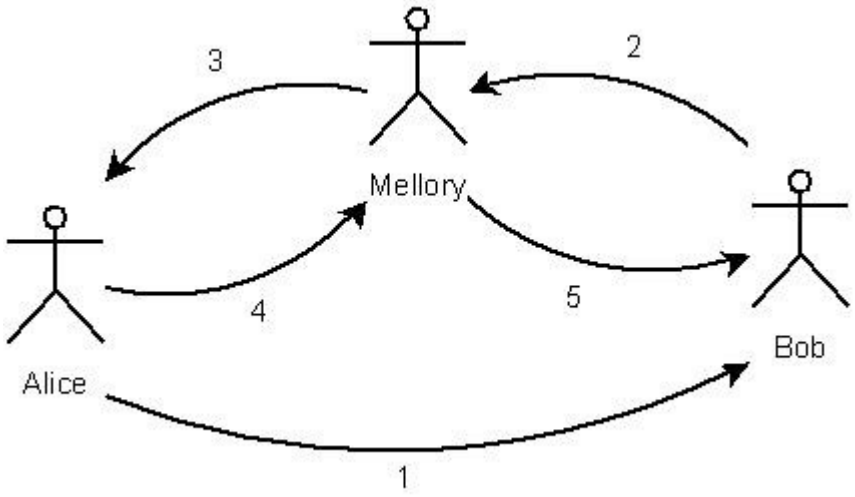
- **Curve Ellittiche** : è una famiglia di algoritmi basati sull'operazione di addizione di curve ellittiche. Questi algoritmi sono usati per la firma, per la cifratura e la gestione delle chiavi.

Man In The Middle

In crittografia, l'attacco dell'uomo in mezzo, meglio conosciuto come *man in the middle attack* o *MITM* è un attacco nel quale l'attaccante è in grado di leggere, inserire o modificare a piacere, messaggi tra due parti senza che nessuna delle due sia in grado di sapere se il collegamento sia stato compromesso. L'attaccante deve essere in grado di osservare e intercettare il transito dei messaggi tra le due vittime.

Ad esempio, supponiamo che Alice voglia comunicare con Bob, e che Mallory voglia spiare la conversazione, e se possibile consegnare a Bob dei falsi messaggi, la comunicazione avviene nella seguente maniera:

1. Alice deve chiedere a Bob la sua chiave pubblica.
2. Se Bob invia la sua chiave pubblica ad Alice, ma Mallory è in grado di intercettarla, può iniziare un attacco Man in the Middle.
3. Mallory può semplicemente inviare ad Alice una chiave pubblica della quale possiede la corrispondente chiave privata.
4. Alice poi, credendo che questa sia la chiave pubblica di Bob, cifra i suoi messaggi con la chiave di Mallory ed invia i suoi messaggi cifrati a Bob.
5. Mallory quindi li intercetta, li decifra, ne tiene una copia per sè, e li re-cifra (dopo averli alterati se lo desidera) usando la chiave pubblica di Bob originariamente inviata ad Alice. Quando Bob riceverà il messaggio cifrato, crederà che questo provenga da Alice.



I principali algoritmi per funzioni di hashing

Di seguito vengono elencati alcuni tra i principali algoritmi a chiave pubblica:

- **MD5 (Message Digest algorithm 5)** : sviluppato da Roland Rivest nel 1991, considera un messaggio di lunghezza arbitraria e produce un riassunto del messaggio (hash) di 128 bit. È l'evoluzione dei precedenti MD2 ed MD4. L'importanza dell'algoritmo è diminuita a causa della compromissione di alcuni dei suoi elementi.
- **SHA1 (Secure Hash Algorithm)** : sviluppato dalla National Security Agency (NSA) e recepito dal NIST in uno standard, produce un riassunto del messaggio di 160 bit. È recepito

anche dall'ISO (International Organization for Standardization).

La firma digitale ed i certificati a chiave pubblica

L'autenticità di molti documenti è determinata dalla presenza di una firma autografata. La firma apposta in calce ad un documento cartaceo esprime, fino a prova contraria, il consenso dell'autore sul contenuto del documento sottoscritto e serve ad attribuirne con certezza la paternità. La crescente diffusione delle tecnologie informatiche in parziale sostituzione delle normali attività lavorative, ha creato un problema non indifferente: quello di trovare un'alternativa alle firme autografe. Nella normale pratica, l'autenticità della firma apposta su un documento cartaceo è verificata attraverso il confronto con un altra firma dello stesso autore, depositata in genere su un documento certificato da un'autorità.

Quello che serve è un metodo col quale ogni corrispondente possa emettere un documento firmato elettronicamente e legalmente riconosciuto. È nata quindi la necessità di individuare un sistema sicuro di scambio di documenti elettronici che abbia la valenza legale di quello tradizionale e che consenta lo sfruttamento delle tecnologie informatiche. Il sistema deve soddisfare i seguenti requisiti:

- paternità del documento digitale deve essere garantita (autenticità);
- il destinatario deve poter verificare l'identità reale del mittente;
- eventuali contraffazioni del messaggio firmato devono essere rilevabili (integrità).

Nell'ambiente informatico i requisiti appena espressi, sono soddisfatti con la *firma digitale* grazie alle tecnologie di crittografia e con l'aderenza alle leggi vigenti, per esempio in Italia l'uso di tale tecnica viene regolata da diverse norme tra cui:

- **Legge 15 marzo 1997 n.59**, detta anche *Legge Bassanini*, nella quale all'art.15 co.2 stabilisce che *"gli atti, dati e documenti formati da pubbliche amministrazioni e dai privati con strumenti informatici o telematici, i contratti stipulati nelle medesime forme, nonché la loro archiviazione e trasmissione con documenti informatici, sono validi e rilevanti a tutti gli effetti di legge";*
- **Decreto del Presidente della Repubblica 10 novembre 1997, n.513**, detto anche *regolamento di attuazione della Legge Bassanini,* ha introdotto il sistema delle firme digitali e stabilisce l'equivalenza tra la firma digitale e la sottoscrizione autografa.
- **Decreto del Presidente del Consiglio dei Ministri 8 febbraio 1999**, contiene le regole tecniche per la formazione, la trasmissione, la conservazione, la duplicazione, la riproduzione e la validazione, anche temporale, dei documenti informatici ai sensi dell'art.3 comma1 del D.P.R.513/97;
- **Decreto del Presidente della Repubblica 28 dicembre 2000, n.445**, detto anche *Testo Unico delle disposizioni legislative e regolamentari in materia di documentazione amministrativa*, il quale ha abrogato la normativa vigente in precedenza, sostanzialmente lasciandola e limitandosi ad una riformulazione letterale ed a una nuova collocazione delle disposizioni.
- **Decreto Legislativo 23 gennaio 2002, n.10**, recepisce la Direttiva 1999/93/CE, relativa alla definizione di un quadro comunitario per le firme elettroniche. Ha introdotto in Italia il concetto di firma elettronica, più generico di quello di firma digitale.
- **Legge n. 48 del 27 febbraio 2008**, pubblicata sulla Gazzetta Ufficiale Serie Generale, n. 80 del 4 aprile 2008 in ratifica ed esecuzione della Convenzione del Consiglio d'Europa sulla criminalità informatica, fatta a Budapest il 23 novembre 2001, e norme di adeguamento

dell'ordinamento interno. Varia ed integra alcuni articoli del Codice Penale tra cui l'articolo **495-bis**. – (Falsa dichiarazione o attestazione al certificatore di firma elettronica sull'identità o su qualità personali proprie o di altri), con il quale si stabilisce che *"chiunque dichiara o attesta falsamente al soggetto che presta servizi di certificazione delle firme elettroniche l'identità o lo stato o altre qualità della propria o dell'altrui persona è punito con la reclusione fino ad un anno"*.

Il meccanismo di generazione di una firma digitale si basa sulla crittografia a chiave pubblica e sulla tecnica di hashing. Il mittente che vuole firmare il proprio messaggio calcola un hash del testo e lo cifra con la propria chiave privata, garantendone quindi la paternità e l'integrità.

La firma digitale è solo concettualmente simile a quella autografa, ma fisicamente molto diversa. Infatti, mentre la firma autografa è sempre la stessa, a meno di imperfezioni non direttamente imputabili alla volontà di chi l'appone, quella digitale dipende da documento firmato e cambia quindi di volta in volta.

In questo scenario informatico manca ancora qualcosa rispetto al mondo dei documenti cartacei dove possiamo attribuire con certezza la firma autografa al suo autore, confrontandola con quella che gli ha depositato presso un notaio o sul proprio documento di identità rilasciatogli dalle autorità pubbliche.

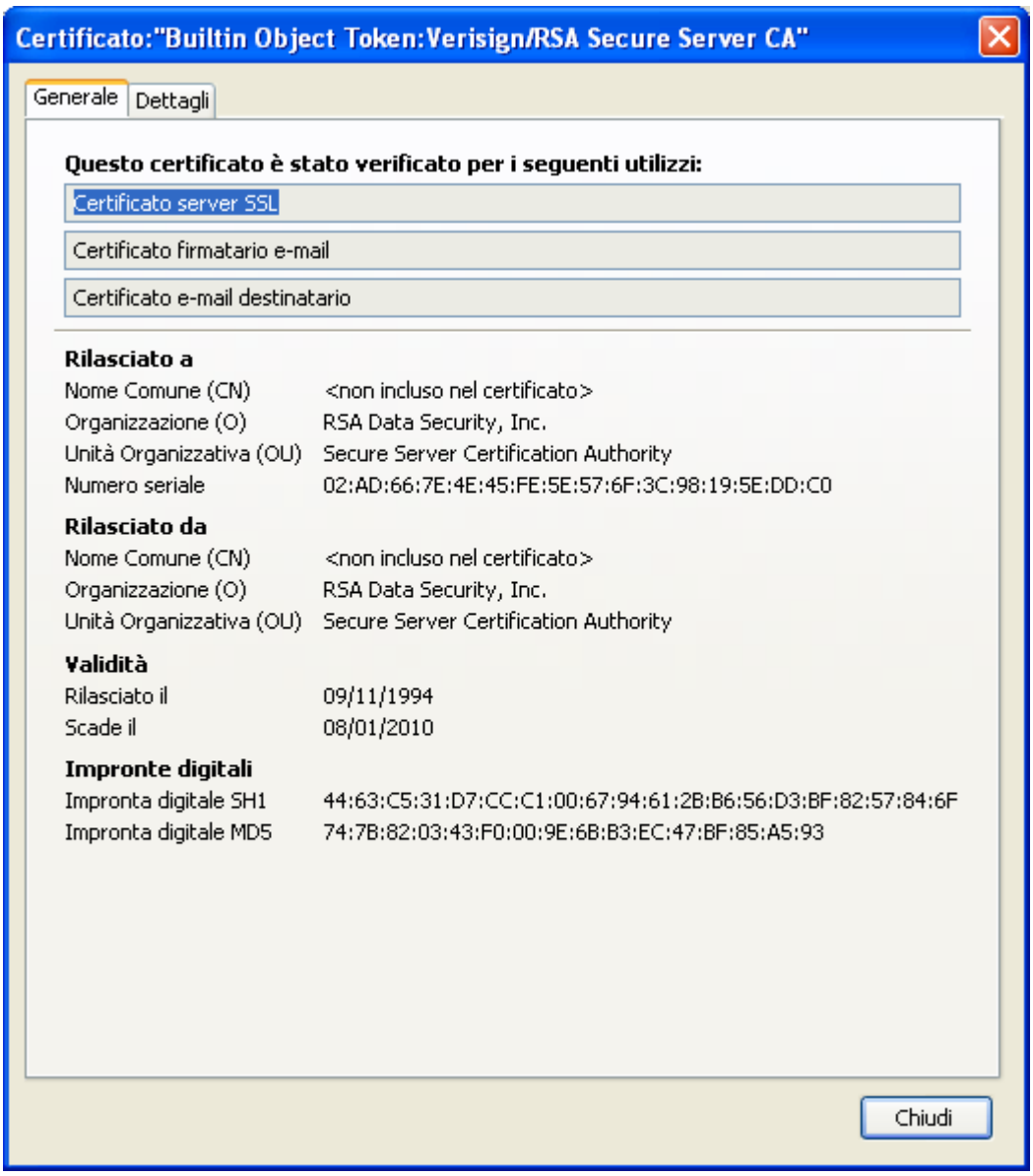
La semplice firma digitale garantisce che un certo messaggio sia stato firmato con una ben precisa chiave privata, ma non assicura che la corrispondente chiave pubblica appartenga alla persona che dice di aver sigillato il messaggio. Poiché una chiave pubblica non è collegata in modo intrinseco ad un soggetto fisico, l'uso del sistema di crittografia a chiave simmetrica permette al destinatario di conoscere il messaggio inviato dal mittente, ma non l'identificazione sicura del mittente stesso. La chiave pubblica, in sostanza, *non ha una corrispondenza fisiologica e incontrovertibile col suo possessore* : questi può dichiararsi e firmarsi secondo un'identità univoca, ma comunque falsa.

Per poter identificare anche fisicamente il mittente di un messaggio, il destinatario ha bisogno di un certificato che ne attesti l'identità. Per questa verifica è necessario l'intervento di una terza parte fidata che abbia la credibilità necessaria a fornire la certezza della corrispondenza tra un soggetto e la sua chiave pubblica. Una terza parte fidata, fatte le verifiche di identità di persona, per esempio con il controllo della carta di identità, emette un certificato in formato elettronico. La soluzione adottata consiste nell'apposizione della firma digitale della terza parte, ovvero cifrare con la sua chiave privata il risultato dell'associazione tra chiave pubblica ed il nome del titolare. Questo risultato costituisce il *certificato a chiave pubblica* o *digitale*. La parte fidata che firma il certificato è chiamata in ambito internazionale *Certification Authority (CA)* ed *Ente certificatore* secondo la normativa italiana.

Il certificato è un documento elettronico che contiene una collezione di informazioni riguardanti il titolare: per esempio informazioni sulla sua identità, il periodo di validità del certificato stesso, l'impronta digitale della chiave pubblica e così via. A queste informazioni viene aggiunta la firma digitale dell'autorità riconosciuta come fidata dalla comunità di utenti all'interno della quale il certificato deve avere circolarità. Esso ha anche una scadenza temporale ed è revocabile sia dall'emettitore sia dall'utente. Lo standard di riferimento dei certificati digitali è l'X.509. Un esempio dei contenuti di un certificato digitale è riportato di seguito:

```
Certificate:
   Data:
       Version: 1 (0x0)
       Serial Number: 7829 (0x1e95)
       Signature Algorithm: md5WithRSAEncryption
       Issuer: C=ZA, ST=Western Cape, L=Cape Town, O=Thawte Consulting cc,
               OU=Certification Services Division,
               CN=Thawte Server CA/emailAddress=server-certs@thawte.com
```

```
        Validity
            Not Before: Jul  9 16:04:02 1998 GMT
            Not After : Jul  9 16:04:02 1999 GMT
        Subject: C=US, ST=Maryland, L=Pasadena, O=Brent Baccala,
                 OU=FreeSoft,
CN=www.freesoft.org/emailAddress=baccala@freesoft.org
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:b4:31:98:0a:c4:bc:62:c1:88:aa:dc:b0:c8:bb:
                    33:35:19:d5:0c:64:b9:3d:41:b2:96:fc:f3:31:e1:
                    66:36:d0:8e:56:12:44:ba:75:eb:e8:1c:9c:5b:66:
                    70:33:52:14:c9:ec:4f:91:51:70:39:de:53:85:17:
                    16:94:6e:ee:f4:d5:6f:d5:ca:b3:47:5e:1b:0c:7b:
                    c5:cc:2b:6b:c1:90:c3:16:31:0d:bf:7a:c7:47:77:
                    8f:a0:21:c7:4c:d0:16:65:00:c1:0f:d7:b8:80:e3:
                    d2:75:6b:c1:ea:9e:5c:5c:ea:7d:c1:a1:10:bc:b8:
                    e8:35:1c:9e:27:52:7e:41:8f
                Exponent: 65537 (0x10001)
    Signature Algorithm: md5WithRSAEncryption
        93:5f:8f:5f:c5:af:bf:0a:ab:a5:6d:fb:24:5f:b6:59:5d:9d:
        92:2e:4a:1b:8b:ac:7d:99:17:5d:cd:19:f6:ad:ef:63:2f:92:
        ab:2f:4b:cf:0a:13:90:ee:2c:0e:43:03:be:f6:ea:8e:9c:67:
        d0:a2:40:03:f7:ef:6a:15:09:79:a9:46:ed:b7:16:1b:41:72:
        0d:19:aa:ad:dd:9a:df:ab:97:50:65:f5:5e:85:a6:ef:19:d1:
        5a:de:9d:ea:63:cd:cb:cc:6d:5d:01:85:b5:6d:c8:f3:d9:f7:
        8f:0e:fc:ba:1f:34:e9:96:6e:6c:cf:f2:ef:9b:bf:de:b5:22:
        68:9f
```



una persona che desidera firmare digitalmente un documento elettronico deve procurarsi un certificato digitale presso una CA riconosciuta internazionalmente o dalla normativa del proprio paese, in particolare in Italia gli enti certificatori sono appositamente censiti nell'Elenco Pubblico dei certificatori tenuto dal CNIPA (Centro Nazionale per l'Informatica nella Pubblica Amministrazione).

Creazione di un certificato in Java

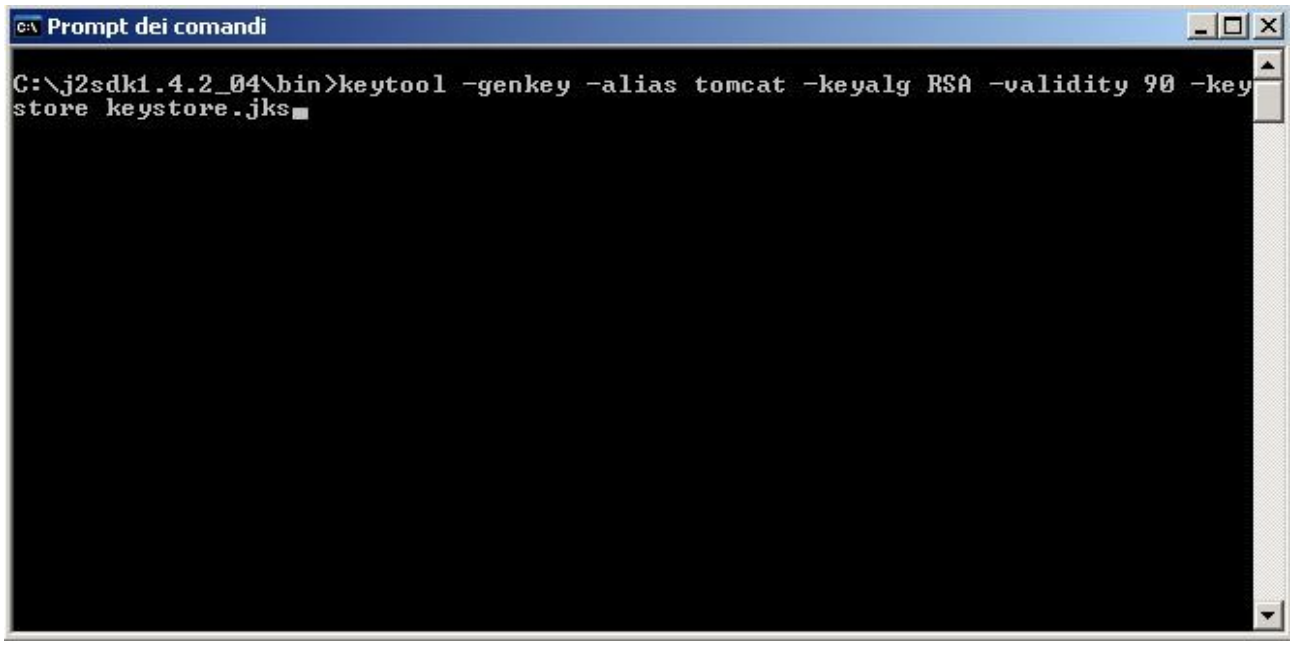
Il programma **keytool** fornito dalla Sun, è un'utility per la gestione di chiavi e certificati digitali. Permette agli utenti di amministrare le loro coppia di chiavi pubblica/privata e di creare i rispettivi certificati per esigenze di auto-autenticazione (dove l'utente identifica se stesso agli altri utenti).

Un certificato digitale è un file che contiene la chiave *pubblica* del sottoscrittore (sia esso una persona fisica o un dispositivo hardware) firmata digitalmente dalla chiave *privata* di una Autorità di Certificazione. Un certificato digitale può essere paragonato ad un passaporto, o ad una carta di identità, cioè ad un documento di riconoscimento rilasciato da un'Autorità universalmente nota, accettata e riconosciuta come affidabile, e utilizzata per autenticare l'identità di una persona.

Il keytool immagazzina le chiavi ed i certificati in un cosiddetto *keystore,* e crea certificati nello standard X.509.

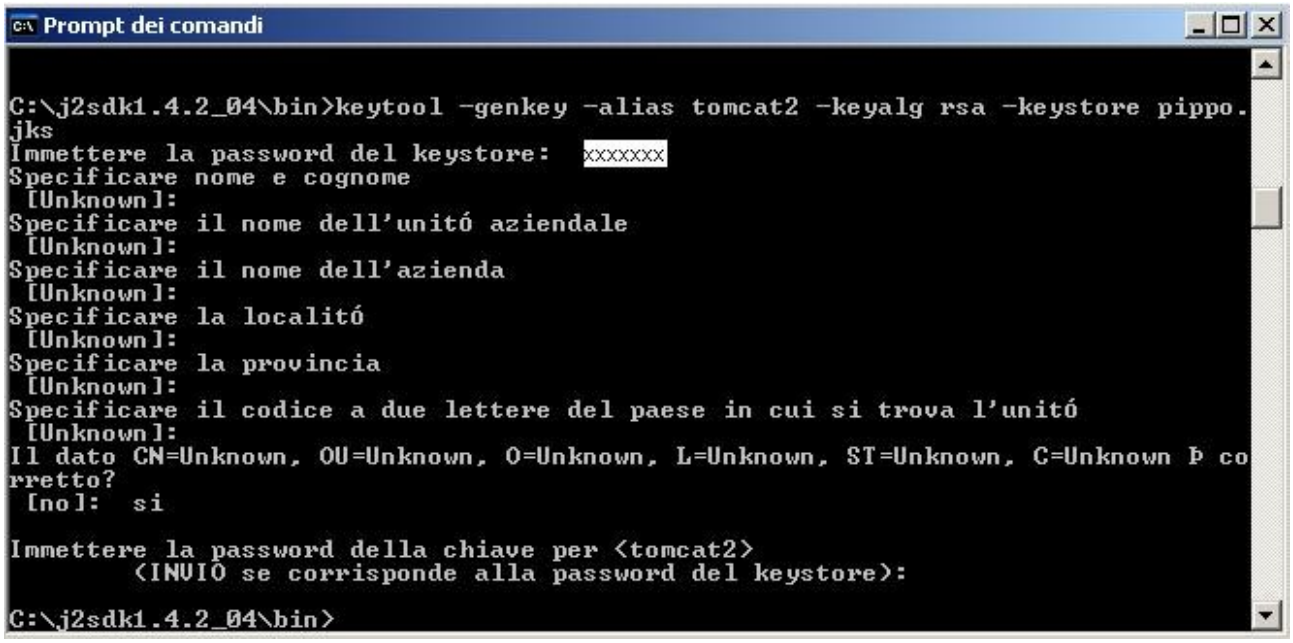
1) **Creare il Keystore**

```
<JAVA_HOME>\bin\keytool -genkey -alias server-alias
-keyalg RSA -validity valDays -keystore keystore.jks
```



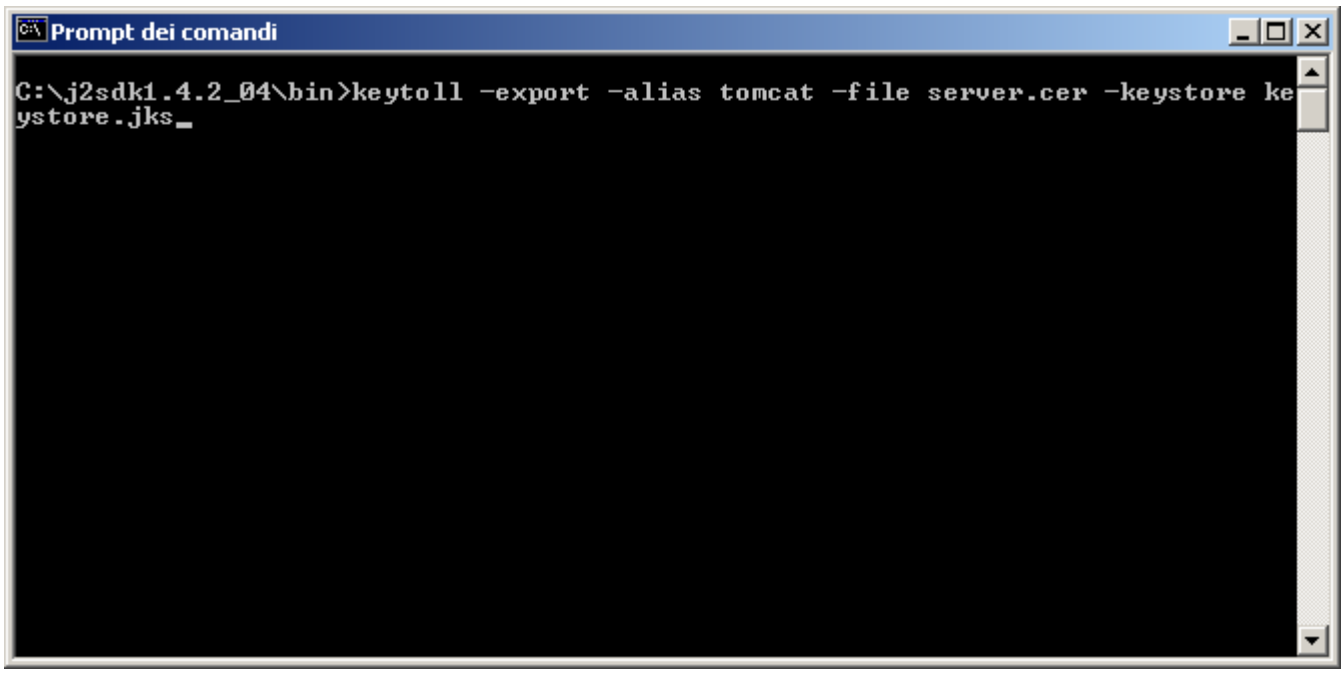
Il programma di utilità "keytool" ci richiederà di inserire le seguenti informazioni:

a. Keystore password -- digitare la password che proteggerà tutto il file(la password di default è "changeit").
b. Nome e Cognome .
c. Unità organizzativa -- immettere il valore adatto, per esempio, Java Software.
d. Organizzazione -- inserirei il valore adatto, per esempio, Sun Microsystems.
e. Città o località.
f. Stato o provincia.
g. Codice a due lettere del paese in cui si trova l'unità aziendale.
h. Password per la chiave privata(premere invio per farla combaciare con la Keystore password).

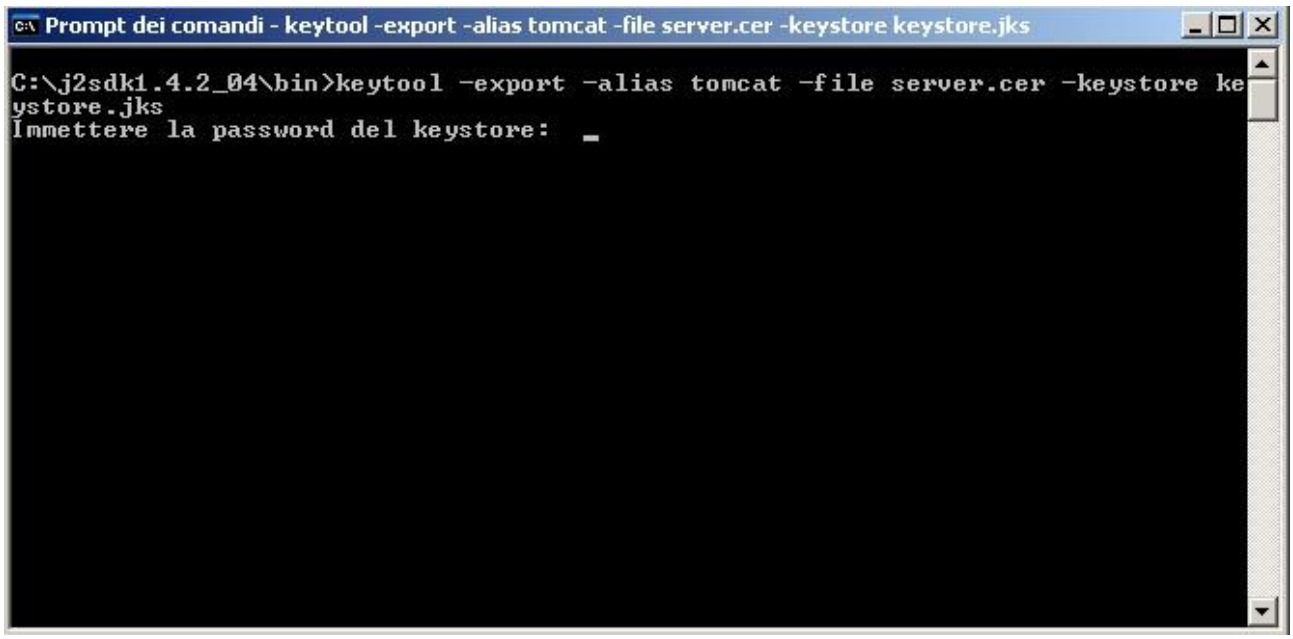


2) **<u>Esportare il certificato del server generato nel file Keystore.jks nel file server.cer</u>**

```
<JAVA_HOME>\bin\keytool -export -alias server-alias -file server.cer -
keystore keystore.jks
```



Nb: Sarà richiesto di immettere la password del "Keystore".



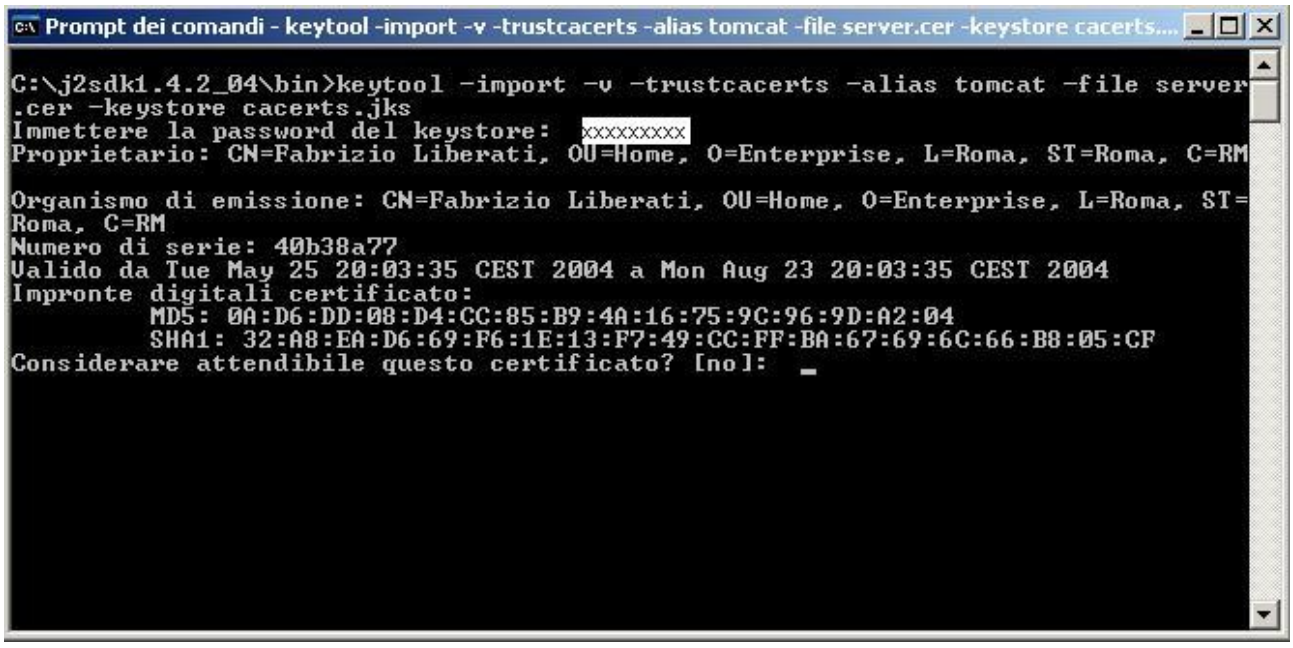
Se la password immessa è corretta, sarà creato il file contente il certificato e sullo schermo apparirà il seguente messaggio di successo.

3) Creare il "trust-store file" cacerts.jks e inserire il certificato del server all'interno del trust-store

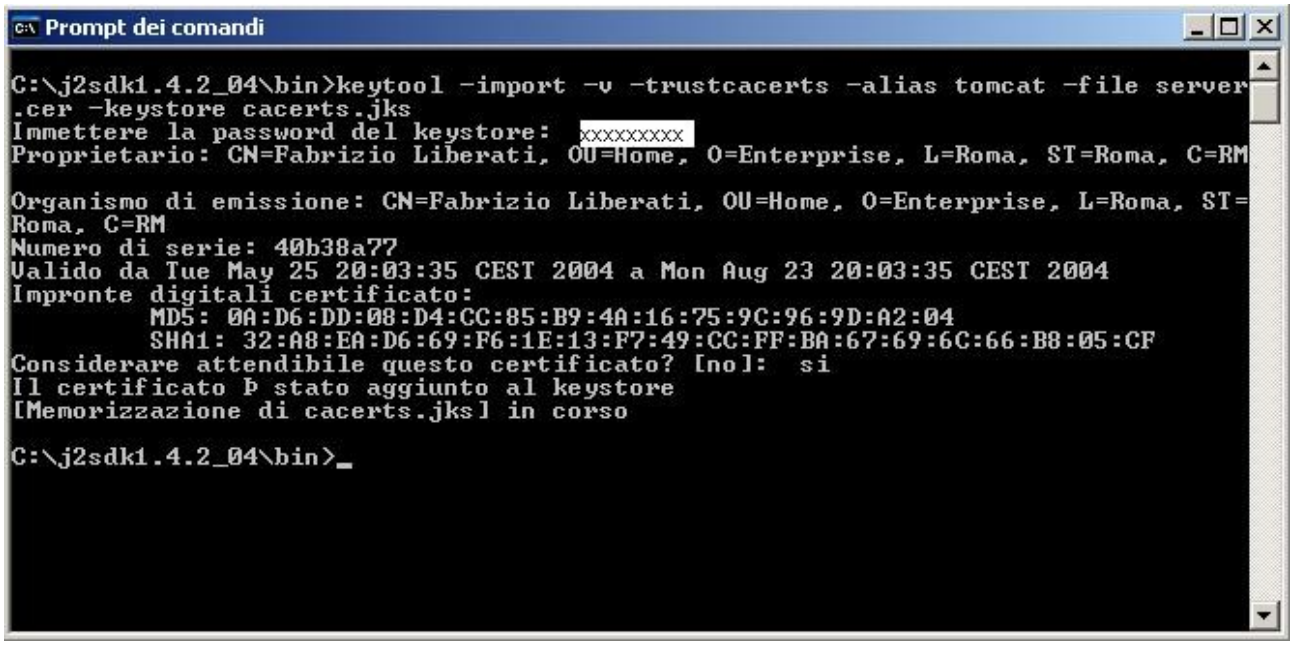
Avviare il Keytool dalla directory dove è stato precedentemente creato il "keystore" e il certificato del server usando i seguenti parametri:

```
<JAVA_HOME>\bin\keytool –import –v –trustcacerts –alias server-alias –file
server.cer –keystore carcets.jks
```

Sarà richiesto di immettere la password del “keystore” e di verificare i dati del certificato.



Se tutto è andato a buon fine apparirà il seguente messaggio:



Le Public Key Infrastructure

Immaginiamo di voler prenotare un albergo via Internet tramite carta di credito. Se vogliamo essere veramente sicuri che la nostra operazione vada a buon fine senza spiacevoli inconvenienti, come ad esempio il ripudio della transazione, dobbiamo spingerci oltre il tradizionale uso dei servizi di autenticazione ed autorizzazione tradizionali. Per garantire che nessuno modifichi i nostri dati che transitano via Internet e per evitare che il fornitore del servizio non rinneghi l'avvenuto pagamento elettronico, le tecniche di crittografia a chiave pubblica, possono fornire una risposta concreta alla nostra esigenza, se integrate con un Ente Certificatore terzo, imparziale tra le parti in causa. Proprio dall'uso delle chiavi pubbliche dei soggetti coinvolti in una comunicazione in un contesto di crittografia asimettrica, prende il nome della struttura che ci affrettiamo a presentare, ovvero la *Public Key* Infrastructure o *PKI*.

Una PKI, o infrastruttura a chiave pubblica, è un insieme di componenti distinti, organizzativi, tecnologici e normativi, i quali interagiscono per consentire agli utenti l'utilizzo semplice e lineare della crittografia a chiave pubblica applicata ad alcune primarie necessità di sicurezza, ovvero l'autenticazione, l'integrità, la riservatezza ed il non ripudio. L'intera infrastruttura è basata sull'uso dei certificati digitali. La PKI rappresenta la risposta alle domande del tipo:

- chi gestisce le chiavi pubbliche e private tipiche della crittografia asimmetrica?
- Chi rilascia o revoca i certificati digitali?
- Chi certifica l'ente che a sua volta rilascia questi certificati?

I certificati digitali sono rilasciati agli utilizzatori da un Ente Certificatore : la *Certification Authority (CA).* Se i certificati sono usati per fini legali, allora tutta l'infrastruttura deve essere a sua volta certificata da un ente governativo preposto, come precedentemente detto in Italia tale ente è il CNIPA. Se vi è un fine legale, un'azienda può creare la propria PKI definendo un ente interno di gestione dei certificati, come del resto fa abitualmente per la gestione dei codici utente, dei profili autorizzativi di accesso alla applicazioni e così via.

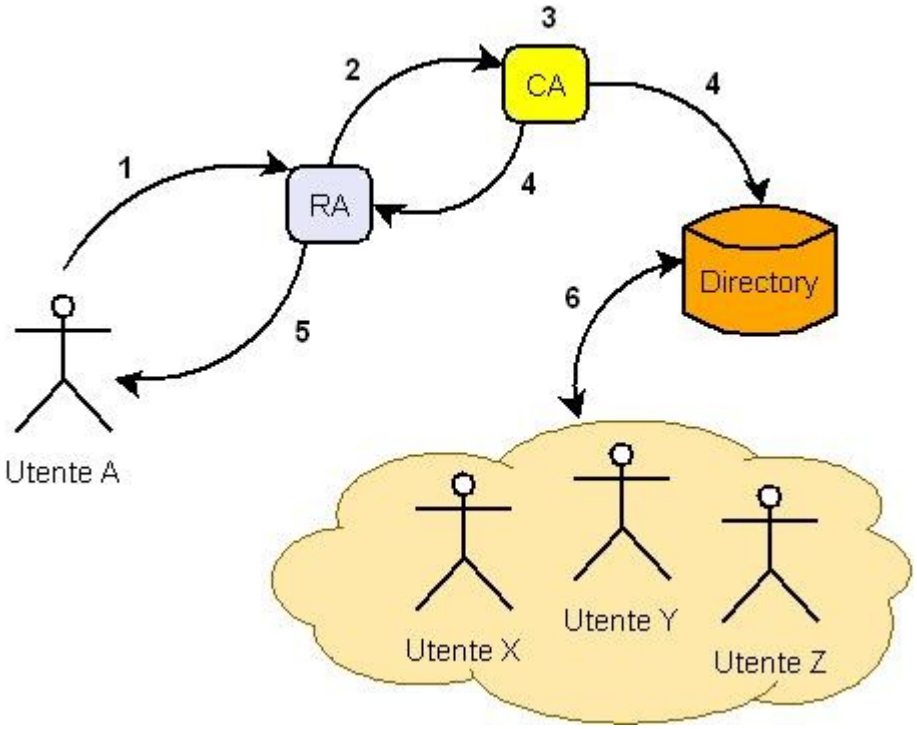
Gli elementi che costituiscono una PKI in genere sono:

- i *certificati digitali* : sono sostanzialmente le chiavi pubbliche degli utenti della PKI, firmate con la chiave privata della CA;

- la *Certification Authority* (*CA*) : ha il compito di gestire i certificati, occupandosi sia della loro emissione sia della loro revoca, attraverso la C*ertificate Revocation List (CRL*), ovvero la lista dei certificati revocati;
- la *Registration Authority (RA)* : ovvero l'autorità di registrazione, richiede i certificati alla propria autorità di certificazione (CA), dopo aver acquisito tutte le informazioni necessarie all'identificazione della persona;
- le *politiche* e le *procedure* : rientrano nel disegno generale delle politiche e procedure di sicurezza dell'azienda e possono essere considerate la loro parte attuativa per quanto riguarda la Certification Authority; riguardano anche le regole per la revoca di un certificato e la relativa emissione di una CRL;
- la C*ertificate Revocation List (CRL*) : ovvero la lista dei certificati revocati, deve essere accessibile dagli utenti della PKI e deve essere mantenuta costantemente aggiornata; oltre alla CRL sono usate anche altre tecniche per la registrazione dei certificati revocati, tra cui l'*On-line certificate Status Protocol (OCSP);*
- il *timestamping* : è una marcatura temporale che serve a garantire la certezza dell'esistenza di un documento elettronico in una certa forma ed in un certo istante;
- la *directory* : è simile ad un database a tecnologia *LDAP (Lightweight Directory Access Protocol)* o X.500, usato per pubblicare i certificati digitali e CRL di una PKI;
- la *cross certification* : è la relazione di mutua fiducia tra differenti CA, ottenuta con lo scambio e il riconoscimento reciproco di certificati emessi da ognuna.

Sostanzialmente il flusso delle operazioni di gestione dei certificati risulta essere il seguente:

1. l'utente richiede il certificato alla RA;
2. la RA verifica l'identità ed inoltra la richiesta alla CA;
3. la CA firma e genera il certificato;
4. la CA inoltra il certificato alla RA e lo registra sulla directory;
5. l'utente riceve il certificato dalla RA;
6. gli utenti accedono alla directory per prelevare i certificati e le CRL.



La crittografia in Internet

Le comunicazioni fisse e mobili stanno evolvendo verso un significativo punto di aggregazione costituito dall'adozione del paradigma dei protocolli Internet. In particolare, le reti basate su TCP/IP (Trasmission Control Protocol e Internet Protocol), costituiscono, oramai, la quasi totalità di quelle per la comunicazione di dati e stanno diventando predominanti anche per reti orientate ad altri flussi informativi, quali la fonia ed il video.

I protocolli Internet, realizzati principalmente negli anni '70 ed '80, hanno considerato in modo abbastanza marginale i problemi di sicurezza derivanti dal loro uso intensivo in un ambiente pubblico popolato da milioni di utenti. Il protocollo TCP/IP, in particolare, non è ritenuto sufficientemente sicuro.

Per ognuno dei livelli previsti dal modello ISO-OSI (Open System Interconnection), la cui mappatura con il TCP/IP è riportata nella figura sottostante, è possibile individuare strumenti di protezione e protocolli standardizzati.

	Modello OSI	**Modello TCP/IP**	
7	Application	Application	*4*
6	Presentation		
5	Session		
4	Transport	Host to Host	*3*
3	Network	Internet	*2*
2	Data Link	Network Acces	*1*
1	Phisical		

Gli strumenti disponibili, come per esempio router, firewall, sistemi di controllo accessi, sono realizzati proprio per migliorare la sicurezza delle reti basate su TCP/IP. Prendiamo ad esempio i firewall: a seconda della loro tipologia possono operare a differenti livelli della cosiddetta pila ISO OSI. Un firewall del tipo *packet filtering*, ad esempio, lavora sia a livello 4 (Transposrt) sia 3 (Network). Un firewall di tipo application level, opera a livello 7 (Application).

Il protocollo TCP lavora a livello 4, l'IP a livello 3, l'HTTP (Hyper Text Transfer Protocol) a livello 7, l'SMTP (Simple Mail Transfer Protocol) sempre a livello 7 e così via. Molti protocolli non sono stati ideati per rispondere alle esigenze attuali di sicurezza. I sistemi crittografici offrono un contributo fondamentale alla sicurezza delle comunicazioni in rete. Dalla posta elettronica ai sistemi di pagamento via Web, dalle reti tradizionali a quelle wireless, la crittografia rappresenta spesso la base per garantire il livello di sicurezza atteso, andando così ad incidere profondamente sul Qualità del Servizio proposto (QoS – Quality of Service). Alcuni protocolli sono stati modificati, in altri casi è stata realizzata l'equivalente versione sicura, ma nella maggior parte dei casi i sistemi di crittografia sono in gran parte utilizzati come valida risposta alle necessità di sicurezza.

Quindi sulla base di tali premesse, a questo punto potremmo giustamente chiederci : a quale livello della pila ISO-OSI vengono applicati i sistemi crittografici? La risposta è : a tutti!

La protezione a livello 1, ovvero quello physical, è applicata al mezzo trasmissivo, per esempio ai cavi, agli amplificatori ai ripetitori ed ai convertitori di segnale. La sicurezza a livello 2, cioè quello data link, si può realizzare con apparati crittografici che cifrano i pacchetti prima di trasmetterli.

A livello 3, cioè a quello network, abbiamo un primo esempio di protocollo che sta alla base di Internet: l'Internet Protocol (IP), che per sua natura non è particolarmente sicuro. Per ovviare alle sue debolezze, sia nell'attuale versione 4 sia nella futura versione 6, è stato realizzato il protocollo IPSec (Internet Protocol Security). Risalendo ancora la pila ISO-OSI, a livello 4, cioè il trasport, abbiamo un esempio il TCP. A livello 7, ovvero l'application, troviamo molti protocolli altrettanto noti, fra i già mensionati HTTP, SMTP e FTP (File Transfer Protocol) e così via.

I protocolli sicuri per Internet

Abbiamo volutamente fornito gli esempi di associazione dei protocolli alla pila ISO-OSI per rammentare a quale livello lavorano e poter introdurre l'argomento dei protocolli sicuri :

- **IPSec** : Specifica un'architettura di sicurezza a livello del protocollo IP. Progettato insieme a Ipv6, ovvero la versione 6 di IP, di cui è parte integrante, può opzionalmente essere utilizzato anche per Ipv4, ovvero la versione attuale, come sua estensione. Consente di realizzare un canale sicuro tra due elementi che comunicano tramite una rete. Può lavorare in *Transport mode*, ovvero solo il corpo del messaggio è cifrato, o in *Tunnel mode*, in cui anche le informazioni di routing e di header sono cifrate. In pratica, IPSec garantisce i requisiti di integrità, autenticità dell'origine, riservatezza e anche il rifiuto di pacchetti trasmessi, tipico di alcuni attacchi.

  IPSec si basa a sua volta su due protocolli denominati rispettivamente:

  - *Authentication Header* : per l'autenticazione e l'integrità dell'header o intestazione del pacchetto;
  - *Encapsulating Security Payload* : per la riservatezza dei dati.

  Per la riservatezza si può ricorrere al DES o triple-DES.

- **SSH (Secure Shell)** : è in grado di creare un canale sicuro tra due computer, detti client SSH e server SSH. Esso permette con semplicità di rendere sicure applicazioni che altrimenti invierebbero informazioni in chiaro su reti pubbliche. Il client ed il server si possono

autenticare reciprocamente tramite uno scambio preliminare di informazioni, mediante un'operazione c.d. di *handshake*. Questo permette anche di stabilire una sessione sicura, sulla quale i dati trasmessi dai due interlocutori viaggiano cifrati. Può essere usato in sostituzione dei servizi Telnet, rlogin, ftp, rexec o rsh.

I protocolli sicuri per la posta elettronica

I protocolli descritti di seguito sono stati sviluppati per garantire la sicurezza delle nostre email:

- **PEM (Privacy Enhanced Mail)** : è uno standard proposto dall'IETF (Internet Engineering Task Force). Usa gli algoritmi di crittografia a chiave sia simmetrica che asimmetrica per garantire riservatezza, integrità ed autenticità della posta via Internet. Per cifrare usa il Triple-DES, nella versione EDE a due chiavi. Per la creazione del codice hash usa gli algoritmi MD2 o MD5. Per la firma e per la distribuzione delle chiavi usa l'algoritmo RSA.
- **S/MIME (Secure Multiporpose Internet Extensions)** : è la versione sicura del protocollo MIME, il quale permette di includere nei normali messaggi di posta elettronica anche file di grafica, audio e altro. S/MIME aggiunge la cifratura e l'uso di firme digitali ai documenti MIME. S/MIME utilizza certificati X.509 e segue la famiglia di standard PKCS (Public Key Cryptography Standards). È basata sull'algoritmo RSA.
- **PGP (Pretty Good Privacy)** : è un programma per lo scambio sicuro di email scritto da Phillip Zimmerman. Esso permette la cifratura delle email tramite l'uso di sistemi di crittografia asimmetrici e simmetrici. Adotta normalmente il sistema RSA per cifrare la chiave segreta, usata a sua volta per cifrare il messaggio, con l'algoritmo IDEA. Le applicazioni più recenti del PGP permettono anche di cifrare file ed interi hard disk.

I protocolli sicuri per il Web

Il World Wide Web è fondamentalmente una ragnatela che avvolge Internet e le reti di tutte le organizzazioni ad essa collegate; è basato sui protocolli TCP/IP e HTTP. Esso permette le più disparate operazioni a tutti coloro che hanno un browser Web installato su un computer. Proprio per questo motivo il Web è una realtà che presenta nuove minacce e vulnerabilità ogni giorno.

Riportiamo alcuni dei protocolli che consentono di ridurre i rischi associati al mondo Web. In generale essi permettono di soddisfare i seguenti requisiti

SSL (Secure Socket layer)

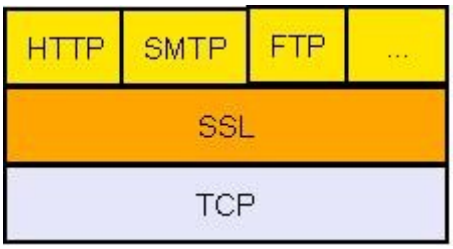
SSL (Secure Socket Layer) fu sviluppato da Netscape Comunications Corp. Esso consente di creare un canale sicuro tra il client ed il server Web. Grazie all'uso di sistemi crittografici, permette l'autenticazione del server e opzionalmente del client, la cifratura dei dati e l'integrità dei messaggi sulla base delle seguenti garanzie:

- **Autenticazione**: l'identità nelle connessioni può essere autenticata usando la crittografia asimmetrica, ovvero a chiave pubblica (RSA, DSS, EL-Gamal). Così ogni client comunica in sicurezza con il corretto server, prevenendo ogni interposizione. È prevista la certificazione del server e, opzionalmente, quella del client.
- **Confidenzialità** o **Riservatezza** nella trasmissione dei dati: la crittografia è usata dopo un *handshake* (accordo o stretta di mano) iniziale per definire una chiave segreta di sessione. In seguito, per crittografare i dati è usata la crittografia simmetrica (AES,3DES, RC4, ecc.).
- **Affidabilità** o **Integrità**: il livello di trasporto include un controllo dell'integrità del messaggio basato su un apposito MAC (*Message Authentication Code*) che utilizza funzioni

hashing (MD5, SHA, RIPEMP-160, ecc). In tal modo si verifica che i dati spediti tra client e server non siano stati alterati durante la trasmissione.

Il protocollo SSL risiede sotto protocolli applicativi quali HTTP, SMTP e NNTP e sopra il protocollo di trasporto TCP e può essere utilizzato per aggiungere sicurezza a qualsiasi protocollo che utilizza TCP, ma il suo utilizzo più comune avviene nel protocollo HTTPS, utilizzato per aggiungere sicurezza alle pagine del World Wide Web in modo tale da rendere possibili applicazioni quali il commercio elettronico.

Per riconoscere l'utilizzo del protocollo HTTPS in una pagina Web, questi utilizza un URL (*Universal Resource Locator*) del tipo *"https://www.sito.xxx"*, dove HTTPS sta per HTTP+SSL. L'SSL opera al di sopra del livello TCP, ovvero il livello 4 del modello ISO-OSI, ed al di sotto dell'HTTP, ovvero il livello 7, come indicato in figura sottostante.



Come detto, per poter instaurare una comunicazione SSL è necessaria una fase preliminare di negoziazione tra le parti coinvolte, denominata *handshake*

Lo scopo dell'handshake è quello di mettere d'accordo client e server sulla terna (ACS, ACA, AC) di algoritmi da utilizzare, dove:

- **ACS** è l'Algoritmo di Crittografia Simmetrico che verrà utilizzato per la comunicazione effettiva;
- **ACA** è l'Algoritmo di Crittografia Asimmetrica che verrà utilizzato per scambiarsi la chiave privata che verrà poi utilizzata nella comunicazione effettiva (crittografata mediante ACS)
- **AC** è l'Algoritmo di Checksum utilizzato per verificare l'integrità dei dati della comunicazione.

L'handshake termina nel momento in cui client e server si sono messi d'accordo sulla tripla di algoritmi da utilizzare. Da questo momento in poi è possibile iniziare la comunicazione effettiva, che verrà crittografata mediante l' ACS scelto, la cui chiave privata è stata scambiata fra le parti mediate l' ACA durante l'handshake.

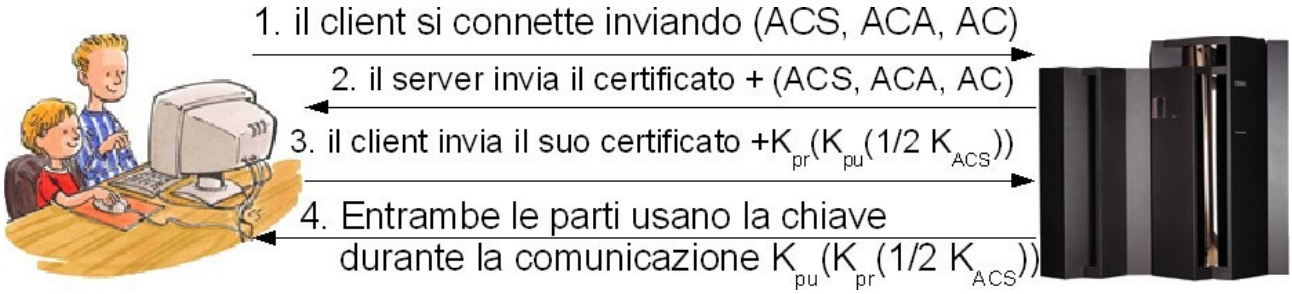


Illustrazione 1: Sessione SSL

1. Il client invia al server le terne (ACS, ACA, AC) che è in grado di utilizzare.
2. Il server risponde con il certificato digitale che ne garantisce l'identità, e il suo set di terne.
3. La prima terna comune a entrambi viene utilizzata per la comunicazione. A questo punto il client spedisce il suo certificato insieme a metà della chiave (generata al momento) dell'ACS

scelto, crittografata con la chiave pubblica del server e, quindi, cripta con la propria chiave privata.

4. Il server spedisce al client l'altra metà della chiave dell'ACS criptata con la propria chiave privata, e quindi criptata con la chiave pubblica del client. Fine dell'handshake.

SHTTP (Securee HTTP)

SHTTP (Secure HTTP) è stato progettato da E. Rescorla e A. Schiffman della EIT (Enterprice Integrated Technologic Inc.), anch'esso per fornire una comunicazione sicura tra client e server in ambito Web ma, mentre le funzionalità di SSL, si applicano all'intera sessione di comunicazione, tra client e server, l'SHTTP si applica a ciascun documento Web. In pratica è un'estensione del protocollo HTTP che permette la trasmissione di pagine Web cifrate, consentendo quindi lo scambio su Internet d'informazioni protette. SHTTP supporta una varietà di algoritmi crittografici, come ad esempio RSA per la firma, DES, IDEA e RC4 per la cifratura, MD5 e SHA-1 per la creazione dell'impronta hash.

I sistemi di pagamento elettronico

Poiché la sicurezza delle trasmissioni via rete è uno degli elementi più critici per la diffusione del commercio elettronico, alcune importanti organizzazioni si sono alleate con l'obiettivo di definire le specifiche per i pagamenti online tramite carta di credito: da questa unione è nata SET (Secure Electronic Transaction). Questi è un protocollo di pagamento per carta di credito, sviluppato da VISA e mastrCard a partire dal 1996, con il supporto di numerosi partner tecnologici. È il risultato della confluenza di due standard inizialmente in competizione fra loro, l'SST (Secure Transaction Technology) di VISA e il SEPP (Secure Elctronic Payment Protocol) di MasterCard. La sua peculiarità è l'interoperabilità: chiunque sviluppi software aderendo a tale standard sarà in grado di scambiare con altri, in modo sicuro, informazioni sui pagamenti elettronici con carte di credito. Per far fronte ai problemi di sicurezza il protocollo SET usa un complesso sistema di autorità di certificazione per verificare l'identità degli acquirenti e dei venditori. Esso garantisce che gli acquirenti ricevano i beni ed i servizi che gli esercenti ricevano il loro pagamento, impedendo l'accesso ai dati della transazione a chi non è autorizzato. Acquirente, ovvero il *cardholder*, esercente, ovvero il *merchant*, banca e distributore della carta sono identificati ciascuno da un certificato digitale. Per le problematiche legate alla privacy, l'architettura del protocollo è stata realizzata in maniera che il merchant abbia accesso solo alle informazioni che definiscono ciò che viene acquistato e non quelle relative ai dati sensibili del pagamento. Similmente, il circuito di autorizzazione ha accesso ai dati relativi al pagamento, ma non alle informazioni sulla merce acquistata. L'utilizzo di SET richiede che acquirente ed esercente dispongano di un apposito software per la gestione della comunicazione; tale software viene normalmente reso disponibile dagli istituti finanziari o dall'esercente stesso. Il SET è basato su algoritmi di cifratura e firma digitale e ha l'obiettivo di garantire:

- la segretezza delle informazioni;
- l'integrità dei dati dei pagamenti;
- l'autenticità del numero di carta di credito (cardholder account);
- l'autenticità del venditore (merchant) e del compratore (cardholder).

Sicurezza e tecnologia wireless

La connettività IP resa disponibile anche da dispositivi wireless rende possibile an'ampia gamma di applicazioni. Queste sono in parte semplici estensioni di alcune d'uso quotidiano, come per esempio

la navigazione in Internet, la consultazione della posta elettronica e così via, ed in parte specifiche del contesto mobile come la localizzazione fisica dell'utente, come ad esempio i navigatori satellitari. La fruibilità di applicazioni basate su IP da terminali mobili ha visto evolversi parallelamente due settori tecnologici: quello della connettività IP per terminali di tipo telefonico cellulare, ovvero telefoni connessi alla rete GSM o GPRS e WAP per la navigazione in Internet, e quello della connettività IP vista come estensione delle reti locali, ovvero come terminali connessi alla rete mediante tecnologia Wireless LAN – WAN. In quest'ultimo caso, la tecnologia ed i protocolli usati sono i medesimi delle reti fisse, a partire dal livello 3 della pila ISO-OSI. La connettività IP in condizioni di mobilità e la conseguente trasmissione nell'etere dei dati scambiati tra elaboratori richiedono un'ulteriore attenzione verso le evidenti problematiche di sicurezza.

L'uso di tecnologie crittografiche sul canale radio è quindi praticamente obbligatoria, sebbene questo aspetto sia spesso trascurato. La tecnologia di questo settore è tuttavia già disponibile. Lo scarso successo riscontrato dal protocollo WEP /Wired Equivalent Privacy), a seguito della provata possibilità di risalire senza eccessiva difficoltà alla chiave utilizzata per cifrare le informazioni, ha accellerato la ricerca di soluzioni alternative quali WPA (Wi-Fi- Protected Access) o il più generale EAP-TLS (Extensible Authentication Protocol – Transport Layer Security). Entrambi sono in grado di garantire all'utente la necessità di riservatezza in rete. Per le tecnologie WLAN basate sui protocolli della classe 802.11 sono stati introdotti criteri di elevata sicurezza utilizzando lo schema di autenticazione fornito dallo standard 802.1x.

BIBLIOGRAFIA

1. La sicurezza dei Sistemi Informativi – Teoria e Pratica a confronto – E. Molteni, F. Faenzi – Mondadori Informatica.
2. Aritmetica, Crittografia e Codici – M.W. Baldoni, C. Ciliberto, G.M. Piacentini Cattaneo – Springer Unitext
3. Dispense del corso di Algebra del Corso di Laurea Specialistica in Informatica dell'Università di Roma – Tor Vergata – Prof.ssa G.M. Piacentini Cattaneo
4. Dispense del corso di Sicurezza Informatica e Cooperazione del Corso di Laurea Specialistica in Informatica dell'Università di Roma – Tor Vergata – Prof. M. Talamo.
5. Dispense del corso di Sicurezza Sistemi Informatici del Corso di Laurea Specialistica in Informatica dell'Università di Roma – Tor Vergata – Prof. F. Arcieri.
6. Dispense del corso di Linguaggi e Protocolli per Sistemi Cooperativi del Corso di Laurea Specialistica in Informatica dell'Università di Roma – Tor Vergata – Prof. A. Dimitri.