Crittografia RSA

Progetto per L'Esame di: Laboratorio di Algoritmi e Strutture di Dati

Edoardo Di Lorenzo 1SI

A.A. 2000-2001

Contents

I Documentazione 3

1 Cos'è la crittografia e a cosa serve? 3
1.1 Crittografia a chiave simmetrica: una chiave . . . 3
1.2 Crittografia a chiave pubblica: due chiavi . . . 3
1.2.1 La chiave . . . 4
1.2.2 Sicurezza . . . 4
1.3 Utilizzo della crittografia a chiave pubblica per le firme digitali. . . . 5
1.3.1 Autenticità . . . 5
1.3.2 Algoritmi hash unidirezionali . . . 6
1.3.3 Firma digitale: combinazione di chiave pubblica e hash . . . 6
1.4 Scambio di chiave: combinazione di chiave simmetrica e chiave pubblica . . . 6

2 Crittosistema RSA 7
2.1 Algoritmo RSA . . . 7
2.2 Cifratura e decifratura . . . 8
2.3 Correttezza . . . 8
2.4 Esempio RSA con piccoli numeri . . . 8
2.4.1 Generazione delle chiavi . . . 8
2.4.2 Cifratura . . . 9
2.4.3 Decifratura . . . 9
2.5 Computazioni di RSA . . . 9

3 Algoritmi utilizzati 9
3.1 Elevazione a potenza modulare . . . 9
3.1.1 Metodo naive . . . 9
3.1.2 Metodo Left-to-Right . . . 9
3.1.3 Metodo Right-to-Left . . . 10
3.1.4 Metodo delle quadrature ripetute . . . 10
3.2 Equazioni modulari . . . 11
3.3 Verifica di primalità . . . 13
3.3.1 Densità dei numeri primi . . . 13
3.3.2 Verifica di pseudoprimalità . . . 14
3.3.3 Verifica di primalità randomizzata di Miller-Rabin . . . 15
3.3.4 Tasso di errore del test di primalità di Miller-Rabin . . . 16

4 Generazione di chiavi 17

5 Sicurezza di RSA 17
5.1 Attacchi RSA . . . 18

6 Storia della crittografia 18

7 Terminologia crittografica 19

II Sorgenti 21

8 Classe: comuni **21**
8.1 comuni.H 21
8.2 comuni.cpp 22

9 Programma: chiavi **27**
9.1 Primi.H 27
9.2 Primi.cpp 27
9.3 Chiavi.H 30
9.4 Chiavi.cpp 31
9.5 genkey.cpp 35

10 Programma: cifra **36**
10.1 Codifica.H 36
10.2 Codifica.cpp 37
10.3 cifrafile.cpp 39

11 Programma: decifra **40**
11.1 decifra.H 40
11.2 decifra.cpp 41
11.3 decodifica.cpp 43

Part I

DOCUMENTAZIONE

1 Cos'è la crittografia e a cosa serve?

Elena e Stefano sono due fidanzati. Stefano vuole spedire una lettera ad Elena ma non vuole che il padre la legga! Come fare? Bisogna trovare un metodo per "manipolare" il messaggio in modo tale che solo Elena possa essere in grado di leggerlo.

La crittografia riveste un'importanza fondamentale per lo scambio sicuro di informazioni sulle reti. Dal punto di vista tecnico, la crittografia è la scienza che si occupa di proteggere i dati trasformandoli matematicamente in un formato non leggibile. Dal punto di vista aziendale, le funzioni di sicurezza fornite dalla crittografia sono:

- l'autenticazione: che assicura al destinatario che l'autore di un messaggio è effettivamente chi dichiara di essere;
- la riservatezza: garantisce che un messaggio venga letto solo dal destinatario desiderato;
- l'integrità: assicura al destinatario che un messaggio non è stato alterato durante il percorso.

1.1 Crittografia a chiave simmetrica: una chiave

La crittografia a chiave simmetrica, chiamata anche crittografia a chiave condivisa o segreta, si basa sull'utilizzo di una singola chiave posseduta sia dal mittente che dal destinatario. Questa chiave, utilizzata sia per la cifratura sia per la decifratura, è denominata chiave segreta, ma è nota anche come chiave simmetrica o chiave di sessione. La crittografia a chiave simmetrica è un modo efficace per crittografare grandi quantità di dati.

Per la crittografia a chiave simmetrica sono disponibili molti algoritmi creati tutti con lo stesso obiettivo, ovvero la trasformazione reversibile di testo normale (dati non crittografati o testo in chiaro) in testo cifrato. Il testo crittografato viene reso incomprensibile utilizzando una chiave di crittografia ed è insignificante per chiunque non disponga della chiave di decifrazione. Con la crittografia a chiave simmetrica viene utilizzata la stessa chiave per la cifratura e la decifratura, quindi la sicurezza di questo processo si basa sull'impossibilità di un utente non autorizzato di ottenere la chiave simmetrica: per questo è chiamata anche crittografia a chiave segreta. Per poter scambiare dati crittografati, gli utenti che desiderano comunicare tramite la crittografia a chiave simmetrica devono prima scambiare la chiave in modo sicuro.

Il fattore determinante per l'efficacia di un algoritmo simmetrico è la dimensione della chiave. Maggiore è la dimensione della chiave, più chiavi devono essere provate per individuare quella che consente di decrittare i dati. Più chiavi sono richieste, più è difficile violare l'algoritmo. Con un algoritmo di crittografia valido e una dimensione della chiave appropriata, è praticamente impossibile per chiunque invertire il processo di trasformazione e ricavare un testo in chiaro dai dati crittografati in un periodo di tempo ragionevole.

1.2 Crittografia a chiave pubblica: due chiavi

Nella crittografia a chiave pubblica vengono utilizzate due chiavi, una pubblica e una privata, matematicamente correlate. Per differenziarla dalla crittografia a chiave simmetrica, la crittografia a chiave pubblica è talvolta chiamata anche crittografia a chiave asimmetrica. Nella crittografia a chiave pubblica la chiave pubblica può essere scambiata apertamente tra le parti o pubblicata in un luogo pubblico, ma la chiave privata rimane segreta. I dati crittografati con la chiave pubblica possono essere decifrati solo utilizzando la chiave privata, mentre quelli crittografati con la chiave privata possono essere decifrati solo tramite la chiave pubblica. Come per la crittografia a chiave simmetrica, anche per quella a chiave pubblica esistono diversi tipi di algoritmi. Tuttavia, gli algoritmi a chiave simmetrica e quelli a chiave pubblica, non sono progettati in modo analogo. È possibile sostituire un algoritmo simmetrico con un altro dello stesso tipo all'interno di un programma, con modifiche minime o senza alcuna modifica poiché funzionano nello stesso modo. Algoritmi a chiave pubblica differenti funzionano invece in modo molto diverso e non sono di conseguenza intercambiabili.

Gli algoritmi a chiave pubblica sono equazioni matematiche complesse che utilizzano numeri molto grandi. La principale limitazione è costituita dal fatto che forniscono forme di crittografia relativamente lente. Di norma vengono utilizzati solo in corrispondenza di punti critici, ad esempio per lo scambio di una chiave simmetrica tra entità o per la firma di un hash di un messaggio. Un hash è un risultato a dimensione fissa ottenuto applicando ai dati una funzione matematica unidirezionale, chiamata algoritmo hash. L'utilizzo di altre forme di crittografia, come la crittografia a chiave simmetrica, unitamente a quella a chiave pubblica, consente di ottenere prestazioni ottimali. La crittografia a chiave pubblica rappresenta un modo efficace per inviare la chiave segreta utilizzata per operazioni di crittografia simmetrica effettuate su grandi quantità di dati. È possibile inoltre utilizzare la crittografia a chiave pubblica insieme ad algoritmi hash per generare una firma digitale.

1.2.1 La chiave

Un generico utente A ha una propria chiave $K_A = (P_A, S_A)$, dove P_A è la parte pubblica e S_A è quella segreta. Le chiavi pubbliche vengono memorizzate in un file a cui tutti gli utenti del crittosistema possono accedere. La chiave pubblica P_A effettua una trasformazione pubblica su un certo messaggio M generando il messaggio $P_A(M)$; mentre la chiave privata P_B effettua una trasformazione privata generando il messaggio $S_A(M)$. Detto M l'insieme dei messaggi, le funzioni $P_A : M \to M$ e $S_A : M \to M$ sono definite, rispettivamente, funzioni di cifratura e decifratura. In tale sistema possiamo avere:

$$M = S_A(P_A(M))$$

$$M = P_A(S_A(M))$$

o entrambe le relazioni.

Uno dei requisiti di queste funzioni è che la funzione di decifratura deve essere del tipo a trabocchetto. Una funzione è detta a trabocchetto quando è facile da calcolare ma difficile da invertire, sempre che non si ha un'informazione segreta aggiuntiva. Così , se $S_A(M)$ è una funzione di questo tipo, la decifratura da parte del crittoanalista è difficile da realizzare. Un esempio di funzione ritenuta a trabochetto è: $f(x) = x^a \ mod \ n$ dove n è il prodotto di due primi p e q abbastanza grandi, a è un intero positivo ed $f : \mathbb{Z}_n \to \mathbb{Z}_n$[1]

1.2.2 Sicurezza

Per consentire la trasmissione di un messaggio da A a B, in sicurezza, le trasformazioni di un sistema a chiave pubblica devono soddisfare $M = S_B(P_B(M))$. Supponendo che A voglia spedire un messaggio M in maniera sicura a B. Allora:

[1] $\mathbb{Z}_n = \{$ -$(n-1)$, ..., -1, 0, 1, 2, ... $n-1$ $\}$

- A ottiene la chiave pubblica di B, P_B;
- A calcola il testo cifrato $C = P_B(M)$ corrispondente al messaggio M e lo spedisce a B;
- B riceve il testo cifrato C e adopera la sua chiave privata S_B per ottenere il messaggio M, $S_B(C) = S_B(P_B(M)) = M$.

Nel caso in cui un nemico intercetti una trasmissione da A, certamente non può essere decifrata perché S_B è privata, per questo motivo la sicurezza è garantita.

$\Rightarrow$ *Claudio è il proprietario di una cantina vinicola che ha cominciato a vendere i propri prodotti via internet. Claudio ha un problema, deve trovare un metodo che gli permetta di verifcare l'identità dell'acquirente in maniera inequivocabile, per essere certo di venire pagato dai propri clienti.*

1.3 Utilizzo della crittografia a chiave pubblica per le firme digitali.

Con una firma digitale gli autori di un messaggio, di un file o di altri dati codificati in modo digitale possono associare la propria identità alle informazioni, ovvero apporvi una firma. L'aggiunta di una firma digitale alle informazioni ne prevede la trasformazione, insieme ad alcuni dati segreti del mittente, in un tag chiamato firma. Per la loro capacità di provare che il mittente di un messaggio è effettivamente chi dichiara di essere e che il messaggio ricevuto è identico a quello inviato, le firme digitali sono utilizzate negli ambienti a chiave pubblica per fornire transazioni sicure per l'e-commerce.

In genere le firme digitali si utilizzano quando i dati vengono diffusi come testo in chiaro, ad esempio con la posta elettronica. In questo caso, anche se la sensibilità del messaggio può non essere tale da giustificare l'uso della crittografia, è comunque importante assicurare che il formato dei dati sia quello originale e che le informazioni non siano state inviate da un impostore.

1.3.1 Autenticità

Per garantire autenticità ed integrità, le trasmissioni di un sistema a chiave pubblica devono soddisfare $P_A(S_A(M)) = M$. Supponiamo che A voglia mandare un messaggio autentico a B. Allora:

- A calcola la sua firma digitale $C = S_A(M)$;
- A spedisce la coppia (M, C) (messaggio, firma);
- Quando B riceverà la coppia (M, C) può verificare che questo provenga da A essendo P_A la chiave pubblica di A, verificando l'equazione $M = P_A(C)$.

Così B può accertare che il messaggio è stato spedito da A e non è stato alterato. Infatti, assumendo che M è un testo in chiaro valido, B sa che C è stato spedito da A e non è stato alterato durante la trasmissione. Questo segue dalla natura a trabocchetto della funzione P_A. Se un crittoanalista parte da un messaggio M dovrebbe, per rompere il sigillo di A, trovare C tale che $P_A(C) = M$, questo implicherebbe che P_A è invertibile, il che si è assunto essere difficile.

1.3.2 Algoritmi hash unidirezionali

Un hash, chiamato anche valore hash o messaggio riassunto (message digest), è un tipo di trasformazione dei dati differente dalla crittografia basata su chiave (simmetrica o pubblica). Un hash è la conversione di un segmento di dati, di qualsiasi lunghezza, in un numero a lunghezza fissa non reversibile tramite l'applicazione ai dati di una funzione matematica unidirezionale, chiamata algoritmo hash. La lunghezza del valore hash risultante è tale da rendere nulla la probabilità che a due segmenti di dati corrisponda lo stesso valore hash. Il mittente genera un hash del messaggio, lo crittografa e lo invia con il messaggio stesso. Il destinatario decifra sia il messaggio sia l'hash, genera un altro hash basato sul messaggio ricevuto e confronta i due hash. Se coincidono, la probabilità che il messaggio trasmesso sia integro è estremamente elevata.

1.3.3 Firma digitale: combinazione di chiave pubblica e hash

È possibile utilizzare la tecnologia a chiave pubblica unitamente agli algoritmi hash per creare una firma digitale, che consenta di verificare l'integrità dei dati e di dimostrare il possesso della chiave privata.

Le operazioni necessarie per la firma e la verifica dei dati eseguite da applicazioni che supportano l'infrastruttura a chiave pubblica sono le seguenti:

- Il mittente applica un algoritmo hash ai dati, generando un valore hash.
- Utilizzando una chiave privata, il mittente trasforma il valore hash in una firma digitale.
- Il mittente invia i dati, la firma e il proprio certificato al destinatario.
- Il destinatario applica l'algoritmo hash ai dati ricevuti e genera un valore hash.
- Il destinatario utilizza la chiave pubblica del firmatario e il valore hash appena generato per verificare la firma.

Gli algoritmi hash sono in grado di elaborare i dati molto più velocemente di quelli a chiave pubblica. Inoltre, l'hashing dei dati riduce la dimensione dei dati da firmare, velocizzando pertanto l'intero processo. Quando si crea o si verifica la firma, l'algoritmo a chiave pubblica deve trasformare solo il valore hash (128 o 160 bit di dati). I dettagli relativi alla creazione e alla verifica della firma dipendono dall'algoritmo a chiave pubblica utilizzato.

1.4 Scambio di chiave: combinazione di chiave simmetrica e chiave pubblica

Gli algoritmi a chiave simmetrica sono eccellenti per crittografare i dati in modo rapido e sicuro. Tuttavia, il loro punto debole è costituito dal fatto che, il mittente e il destinatario devono scambiarsi una chiave segreta prima di poter scambiare i dati. L'utilizzo combinato di algoritmi simmetrici per crittografare i dati e di algoritmi a chiave pubblica per scambiare la chiave segreta rappresenta quindi una soluzione veloce e al tempo stesso scalabile.

Le operazioni necessarie per uno scambio di chiave basato su chiave pubblica sono le seguenti:

- Il mittente ottiene la chiave pubblica del destinatario.
- Il mittente crea una chiave segreta casuale (la singola chiave utilizzata nella crittografia a chiave simmetrica).

- Il mittente utilizza la chiave segreta con un algoritmo simmetrico per convertire il testo normale in testo crittografato.
- Il mittente utilizza la chiave pubblica del destinatario per trasformare la chiave segreta in una chiave segreta crittografata.
- Il mittente invia il testo e la chiave segreta crittografati al destinatario.
- Il destinatario converte la chiave segreta crittografata in testo normale utilizzando la propria chiave privata.
- Il destinatario converte il testo crittografato in testo normale utilizzando la chiave segreta non crittografata.

2 Crittosistema RSA

Nel 1978 Rivest, Shamir e Adleman pubblicarono sulla rivista "Communications of the ACM" la descrizione del cifrario RSA (dalle loro iniziali) che possedeva molte interessanti caratteristiche: costituiva (insieme al cifrario di Merkle ed Hellman) il primo esempio di cifrario a chiave pubblica; operava a blocchi interpretando ogni messaggio come un numero intero; la sua sicurezza si fondava sulla presunta difficoltà computazionale della fattorizzazione di numeri interi molto grandi. Sebbene abbia attratto l'interesse di numerosi crittoanalisti, la RSA rimane sino ad oggi sostanzialmente inviolata, e se a tutto ciò si aggiunge una grandissima semplicità strutturale, ecco che possiamo ben capire perché questo metodo di cifratura abbia avuto così tanto successo. Per la sua semplicità la RSA è ampiamente utilizzata e numerose sono le sue realizzazioni in hardware presentate nel corso di questi anni. Si è detto che la sicurezza della RSA sia strettamente legata alla difficoltà del problema della fattorizzazione, ma nessuno ha mai provato che questo legame significhi che i due problemi siano computazionalmente equivalenti. Comunque tutti gli sforzi fatti per rompere l'RSA hanno portato inevitabilmente ad individuare algoritmi per la fattorizzazione. D'altro canto, esistono delle varianti di RSA (Rabin, 1979) che sono dimostrativamente equivalenti al problema della fattorizzazione, e dunque ritenute più sicure almeno da un punto di vista matematico. Però queste varianti non sono semplici come l'RSA, e quindi non hanno avuto il suo stesso successo.

2.1 Algoritmo RSA

Generazione delle chiavi

1. A genera due numeri primi grandi p e q;
2. A calcola $n = p \cdot q$ e $\Phi(n) = (p-1)(q-1)$;
3. A sceglie un numero $1 < e < \Phi(n)$ tale che $gcd(e, \Phi(n)) = 1$;
4. A calcola $d = e^{-1}\ mod\ \Phi(n)$[2];
5. A pubblica n ed e come sua chiave pubblica $P_A = (e, n)$;
6. A conserva n e d come sua chiave privata $S_A = (d, n)$.

[2] Dato un numero naturale n si indica con $\Phi(n)$ il numero di naturali minori o uguali a n e coprimi con n. La funzione Φ si chiama *funzione* Φ *diEulero*.

Per generare la chiave pubblica di cifratura, l'utente A del sistema RSA sceglie a caso due numeri primi piuttosto grandi p e q; il loro prodotto $n = p \cdot q$, unito ad un numero e, costituiscono la chiave pubblica $P_A = (e, n)$ del sistema. Il numero e è scelto in modo che $gcd(e, \Phi(n)) = 1$ dove $\Phi(n) = (p-1)(q-1)$, è la funzione di Eulero. La chiave privata è costituita dalla coppia $S_A = (d, n)$, dove d è calcolato come l'inverso moltiplicativo di $e \ mod \ \Phi(n)$. La proprietà che e e $\Phi(n)$ sono coprimi garantisce sempre l'esistenza dell'inverso moltiplicativo d.

2.2 Cifratura e decifratura

Per effettuare la cifratura di un messaggio $M \in \mathbb{Z}_n$, il mittente A preleva dall'elenco la chiave pubblica P_B dell'utente destinatario B e calcola il testo cifrato

$$C = P_B(M) = M^e \ mod \ n$$

Il destinatario conoscendo la sua chiave segreta $S_B = (d, n)$ calcola il messaggio in chiaro

$$M = S_B(C) = C^d \ mod \ n$$

2.3 Correttezza

Verifichiamo ora che la fase di cifratura e quella di decifratura sono una l'inversa dell'altra. Poiché

$$ed \equiv 1 \ mod \ \Phi(n)$$

si ha

$$ed = t \cdot \Phi(n) + 1$$

con t intero e $t \geq 1$. Supposto che $M \in \mathbb{Z}_n^*$[3], si ha:

$$C^d \ mod \ n = (M^e \ mod \ n)^d \ mod \ n = M^{t \cdot \Phi(n)+1} \ mod \ n = M$$

in quanto per il teorema di Eulero si ha che: $x^{\Phi(n)} = 1 \ mod \ n$. Nel caso in cui la lunghezza di M sia maggiore di n, si avrebbe che M e $(M \ mod \ n)$ originerebbero lo stesso crittogramma. Per ovviare a questo problema si assume che i messaggi siano numeri minori di n. Questo lo si può semplicemente ottenere decomponendo la sequenza di bit rappresentanti il messaggio in chiaro, in gruppi di $l = log_2 \ n$ bit ciascuno. Così facendo si garantisce che $m < 2^l < n$.
Nel caso in cui $M \in \mathbb{Z}_n - \mathbb{Z}_n^*$ per la correttezza si utilizza il teorema del resto cinese.

2.4 Esempio RSA con piccoli numeri

2.4.1 Generazione delle chiavi

- A sceglie due numeri primi p e q: $p = 47$, $q = 71$;
- A calcola $n = p \cdot q$ e $\Phi(n) = (p-1) \cdot (q-1)$
  $n = 47 \cdot 71 = 3337$, $\Phi(n) = 46 \cdot 70 = 3220$
  A sceglie e tale che $gcd(e, \Phi(n)) = 1$ $e = 79$
  A calcola $d = e^{-1} \ mod \ \Phi(n)$ $d = 79^{-1} \ mod \ 3220 = 1019$
  La chiave pubblica è la coppia $(e, n) = (79, 3337)$
  La chiave privata è la coppia $(d, n) = (1019, 3337)$

[3] L'insieme degli elementi invertibili di $\mathbb{Z}_n$ si indica con $\mathbb{Z}_n^*$. Per ogni $n > 0$, si ha che $|\mathbb{Z}_n^*| = \Phi(n)$. Sia $u \in \mathbb{Z}_n^*$ allora $u^{\Phi(u)} = 1$ in $(\mathbb{Z}_n)$.

2.4.2 Cifratura

- Un utente B spedisce ad A il messaggio $M = 688$ conoscendo la sua chiave pubblica $P_A = (79, 3337)$
- B calcola $C = M^e \ mod\ n \quad C = 688^{79} \ mod\ 3337 = 1570$

2.4.3 Decifratura

- A riceve il messaggio cifrato $C = 1570$
- A si ricava M con la formula $M = C^d \ mod\ n$
  $M = 1570^{1019} \ mod\ 3337 = 688$

2.5 Computazioni di RSA

Le procedure del crittosistema RSA si basano sull'elevazione a potenza modulare e sulla risoluzione di equazioni modulari.

3 Algoritmi utilizzati

3.1 Elevazione a potenza modulare

L'elevamento a potenza modulare è un'operazione essenziale in molte procedure di verifica di primalità e nel sistema di crittografia a chiave pubblica RSA.

L'operazione dell'elevazione a potenza modulare consiste nel calcolare $x^y \ mod\ z$ dove x, y e z sono interi. Presentiamo varie procedure per effettuare il calcolo.

3.1.1 Metodo naive

Intuitivamente si può semplicemente moltiplicare x per se stesso y volte facendo ad ogni passo modulo z per evitare cifre troppo grandi. Tale metodo è alla base del seguente algoritmo:

```
POTENZA MODULARE NAIVE(x,y,z)
  a ← 1;
  for i = 1 to y do
    a ← (a·x) mod z
  return a
```

Tale algoritmo in realtà non è utilizzato in quanto non efficiente. Il numero cicli, infatti, è uguale a y, ossia esponenzialmente nella lunghezza dell'encoding (binario) di y. Quindi se y è di 512 bit si avranno 2^{512} cicli. I metodi realmente utilizzati sono tre: il metodo *Left to right*, il metodo *Right to left* e il metodo delle *quadrature ripetute*[4]".

3.1.2 Metodo Left-to-Right

Ricordando che vogliamo calcolare $x^y \ mod\ z$, supponiamo che $y_0 y_1 y_2 y_3 ... y_t$ siano i bit da cui è formato y espresso in base due, il valore di y è quindi:

$$y = y_0 + 2(y_1 + 2(y_2 + 2(y_3 ... 2(y_{t-1} + 2y_t)...)))$$

[4] è il metodo che ho usato io

da cui

$$x^y = x^{y_0}(...(x^{y_{t-1}}(x^{y_t})^2)^2...)$$

Da tali considerazioni si ottiene il seguente algoritmo:

```
METODO LEFT TO RIGHT(x,y,z)
   a ← 1;
   for i = t downto 0 do
      a ← (a·a) mod z
      if y_i = 1 then a ← (a·x) mod z
   return a
```

Chiaramente tale algoritmo è migliore rispetto a quello precedente infatti se y è composto da 512 bit ora il numero di cicli è 512.

3.1.3 Metodo Right-to-Left

Anche tale metodo segue da considerazioni matematiche infatti si ha che:

$$y = y_0 2^0 + y_1 2^1 + ... + y_{t-1} 2^{t-1} + y_t 2^t$$

da cui

$$x^y = (x^{2^0})^{y_0}(x^{2^1})^{y_1}...(x^{2^t})^{y_t}$$

Da ciò segue l'algoritmo per il calcolo di $x^y \ mod \ z$:

```
METODO RIGHT TO LEFT(x,y,z)
   if y = 0 then return 1
   X ← x; P← 1;
   if y_0 = 1 then X ← x
   for i = 1 to t do
      X ← (X·X) mod z
      if y_i = 1 then P ← (P·X) mod z
   return P
```

Anche questo metodo impiega 512 cicli se y è di 512 bit.

3.1.4 Metodo delle quadrature ripetute

Il metodo delle quadrature ripetute risolve il problema dell'elevazione a potenza modulare efficientemente usando la rappresentazione binaria di y. Sia $(y_k, y_{k-1}, ..., y_1, y_0)$ la rappresentazione binaria di y. (Cioé, la rappresentazione binaria è lunga $k+1$ bit, y_k è il bit più significativo e y_0 è il meno significativo.) La seguente procedura calcola $x^c \ mod \ z$, aumentando c attraverso raddoppi e incrementi da 0 a y.

```
MODULAR-EXPONENTIATION(x,y,z)
   c ← 0
   d ← 1
```

```
sia (y_k, y_{k-1}, ..., y_1, y_0) la rappresentazione binaria di y
for i ← k downto 0
   do c ← 2c
      d ← (d·d) mod z
      if y_i = 1
         then c ← c+1
         d ← (d·x) mod z
return d
```

Ogni esponente calcolato in una sequenza è il doppio del precedente esponente oppure il precedente esponente incrementato di 1; la rappresentazione binaria di y è letta da destra a sinistra per controllare quali operazioni sono eseguite. Ogni iterazione del ciclo usa una delle identità

$$x^{2c}\ mod\ z = (x^c)^2\ mod\ z$$

$$x^{2c+1}\ mod\ z = x \cdot (x^c)^2\ mod\ z$$

a seconda che $y_i = 0$ o 1, rispettivamente. L'uso essenziale del quadrato ad ogni iterazione spiega il nome "quadrature ripetute". Appena il bit y_i è letto e elaborato, il valore di c è lo stesso del prefisso $(y_k, y_{k-1}, ..., y_i)$ della rappresentazione binaria di y. Per esempio, per $x = 7$, $y = 560$ e $z = 561$, l'algoritmo calcola la sequenza dei valori modulo 561 mostrata nella tabella; la sequenza di esponenti usati è mostrata nella riga c della tabella. La variabile c non è in realtà necessaria nell'algoritmo, ma è stata aggiunta per poter dare una spiegazione migliore: l'algoritmo mantiene l'invariante che $d = x^c\ mod\ z$ al crescere di c con raddoppi e incrementi finché $c = y$. Se gli input x, y e z sono numeri di β bit, allora è richiesto un numero totale $O(\beta^3)$ di operazioni sui bit.

i	9	8	7	6	5	4	3	2	1	0
y_i	1	0	0	0	1	0	0	0	0	0
c	1	2	4	8	17	35	70	140	280	560
d	7	49	157	526	160	241	298	166	67	1

Table 1: x=7, y=560=(1000110000) e z=516. I valori sono mostrati dopo ogni esecuzione del ciclo for. Il risultato finale è 1.

3.2 Equazioni modulari

Le equazioni modulari consistono nel risolvere l'equazione $ax \equiv b\ (mod\ n)$. Tale calcolo richiede la conoscenza dell'algoritmo di Euclide e dell'algoritmo di Euclide Esteso.

Algoritmo di Euclide

L'algoritmo di Euclide, descritto negli "Elementi di Euclide" [300 a.C.], è utilizzato principalmente per il calcolo del Massimo Comune Divisore; l'algoritmo di Euclide esteso è utilizzato per la risoluzione delle equazioni lineari modulari. Il nostro interesse a risolvere tali equazioni dipende dal fatto che sono utilizzate per il calcolo di un parametro fondamentale nel sistema crittografico RSA. L'algoritmo di Euclide si basa sul "Teorema di ricorsione del Massimo Comune Divisore" che dice: "Per tutti gli interi $a \geq 0$ e $b > 0$ il $gcd(a, b)$ è uguale al $gcd(b, a\ mod\ b)$".

```
EUCLIDE(a,b)
   if b = 0 then return a
```

```
    else return EUCLIDE (b, a mod b)
```

Assumendo $a > b$ si ha che l'algoritmo di Euclide esegue al più $\log b$ chiamate, in ogni chiamata si hanno $O(\log^2 a)$ operazioni sui bit da cui in totale si hanno $O(\log^3 a)$ operazionisui bit. Da un'analisi pù accurata in realtà si ottiene che Euclide richiede al più $O(\log^2 a)$ operazioni sui bit.

Esempio: consideriamo la computazione del gcd(30, 21):

$$EUCLIDE(30, 21) = EUCLIDE(21, 9) = EUCLIDE(9, 3) = EUCLIDE(3, 0) = 3$$

In questa computazione ci sono solo tre chiamate ricorsive.

L'algoritmo non sarà ricorsivo all'infinito poiché il secondo parametro decresce strettamente in ogni chiamata ricorsiva. Così l'algoritmo di Euclide termina sempre con una risposta corretta.

Algoritmo di Euclide Esteso

La procedura prende in input due interi arbitrari (a e b) e restituisce una tripla (g, x, y) che soddisfa l'equazione:

$$g = gcd(a, b) = ax + by$$

Il seguente pseudo-codice implementa l'algoritmo di Euclide Esteso:

```
EUCLIDE-ESTESO(a, b)
    if b = 0 then return (a, 1, 0)
    (g', x' ,y') ← EUCLIDE-ESTESO (b, a mod b)
    (g, x, y) ← (g', y', x'-(a/b)·y')
    return (g, x, y)
```

Quest'algoritmo permette di risolvere equazioni del tipo $ax \equiv b \ (mod\ n)$ o $ax \equiv 1 \ (mod\ n)$. Tale algoritmo ha un tempo d'esecuzione uguale all'algoritmo di Euclide.

Esempio EUCLIDE-ESTESO(99, 78):

a	b	(a/b)	g	x	y
99	78	1	3	-11	14
78	21	3	3	3	-11
21	15	1	3	-2	3
15	6	2	3	1	-2
6	3	2	3	0	1
3	0	–	3	1	0

Table 2: EUCLIDE-ESTESO(99, 78)

Da cui i valori finali: $g = 3$, $x = -11$ e $y = 14$.
Possiamo dimostrare che tali valori sono corretti sostituendo tali valori in $g = gcd(a, b) = ax + by$. Quindi abbiamo che $3 = 99 \cdot (-11) + 78 \cdot 14$ come volevasi dimostrare.

Risoluzioni di equazioni modulari

Supponiamo di voler risolvere l'equazione $ax \equiv b\ (mod\ n)$, tale equazione ha soluzione se e solo se $g|b$, dove $g = gcd(a, n)$. In tal caso ci sono esattamente g distinte soluzioni modulo n:

$$x'(b/g) + i(n/g) \text{ per } i = 0, 1, ...g - 1$$

dove x' si ottiene risolvendo l'equazione $g = ax' + ny$ utilizzando l'algoritmo EUCLIDE-ESTESO(a, n) precedentemente trattato.
Il tipo di equazioni lineari modulari utilizzate nella RSA è $ax \equiv 1\ (mod\ n)$, essa ha soluzione se e solo se $gcd(a, n) = 1$.
La soluzione è x' dove $1 = ax' + ny$.
Usando EUCLIDE-ESTESO(a, n) è possibile ottenere tale soluzione, che viene denotata con $a^{-1}\ mod\ n$.

3.3 Verifica di primalità

3.3.1 Densità dei numeri primi

Per molte applicazioni di crittografia vi è la necessità di trovare primi grandi "casuali". Fortunatamente, i primi grandi non sono troppo rari, così che non è richiesto troppo tempo per trovare un primo verificando via via interi casuali della dimensione appropriata. La funzione di distribuzione dei primi $\pi(n)$ specifica il numero di primi che siano minori o uguali a n. Per esempio $\pi(10) = 4$, poiché vi sono 4 numeri primi minori o uguali a 10, cioé 2, 3, 5 e 7. Il teorema del numero primo fornisce un'utile approssimazione di $\pi(n)$.

Teorema 3.1 *(Teorema del numero primo)*

$$\lim_{n\to\infty} \frac{\pi(n)}{n/\ln n} = 1$$

L'approssimazione $n/\ln n$ fornisce una stima ragionevolmente accurata di $\pi(n)$ anche per piccoli n. Per esempio, dà un errore per meno del 6% per $n = 109$, dove $\pi(n) = 50847478$ e $n/\ln n = 48254942$. (Per un teorico dei numeri, 10^9 è un numero piccolo.)

Si può usare il teorema del numero primo per stimare a $1/\ln n$ la probabilità che un intero n scelto casualmente sia primo. Pertanto, si dovrebbero esaminare approssimativamente $\ln n$ interi scelti casualmente vicino ad n per trovare un primo che sia della stessa lunghezza di n. Per esempio, trovare un primo di 100 cifre potrebbe richiedere la verifica di primalità di approssimativamente, $\ln 10^{100} \approx 230$ numeri di 100 cifre scelti casualmente. (Questo numero può essere diviso a metà scegliendo solo interi dispari.)

Nel resto del paragrafo, si considera il problema di determinare se un intero n dispari grande è primo. Per comodità di notazione, si assuma che n abbia la scomposizione in fattori

$$n = p_1^{e_1}\ p_2^{e_2}\ ...\ p_r^{e_r}$$

dove $r \geq 1$ e $p_1, p_2, ..., p_r$ sono i fattori primi di n. Naturalmente, n è primo se e solo se $r = 1$ e $e_1 = 1$. Un semplice approccio al problema di verificare la primalità è la prova di divisione. Si cerca di dividere n per ogni intero $2, 3, ..., \sqrt{n}$. (Come prima, interi pari più grandi di 2 possono essere saltati.) è facile vedere che n è primo se e solo se nessuno dei divisori provati divide n. Assumendo che ogni prova di divisione richieda tempo costante, il tempo di esecuzione è $\Theta(\sqrt{n})$ nel caso peggiore, che è esponenziale rispetto alla lunghezza di n. (Si ricorda che se n è rappresentato in binario usando β bit, allora $\beta = \log(n + 1)$, e così $\sqrt{n} = \Theta(2^{\beta/2})$).

Pertanto, la prova di divisione funziona bene solo se n è molto piccolo oppure se ha un fattore primo piccolo. Quando funziona, la prova di divisione offre il vantaggio non solo di determinare se n è primo o composto, ma, se n è composto, di determinarne di fatto anche la scomposizione in fattori primi.

In questo paragrafo, si è interessati a conoscere solo se un dato numero n è primo; se n è composto, non interessa scomporlo in fattori primi. Il calcolo della scomposizione di un numero in fattori primi, da un punto di vista computazionale, è costoso. Può sorprendere forse il fatto che è molto più facile dire se un dato numero è primo o no che determinare la sua scomposizione in fattori primi se non è primo.

3.3.2 Verifica di pseudoprimalità

Si consideri ora un metodo per il controllo della primalità che è "quasi corretto" e di fatto sufficientemente buono in molte applicazioni pratiche. Una raffinamento di questo metodo che ne rimuove i piccoli difetti sarà presentato in seguito. Si denotino con $\mathbb{Z}_n^+$ gli elementi non nulli di $\mathbb{Z}_n$: $\mathbb{Z}_n^+ = \{1, 2, ..., n-1\}$.
Se n è primo, allora $\mathbb{Z}_n^+ = \mathbb{Z}_n^*$. Si dice che n è uno pseudoprimo di base a se n è composto e $a^{n-1} \equiv 1 \ (mod\ n)$.

Teorema 3.2 *(Teorema di Fermat)*
Se p è primo, allora, $a^{p-1} \equiv 1 \ (mod\ p)$ per $a \in \mathbb{Z}_p^*$.

Il teorema di Fermat implica che se n è primo, allora n soddisfa l'equazione a^{n-1} $(mod\ n)$ per ogni a in $\mathbb{Z}_n^+$. Pertanto, se si può trovare qualche $a \in \mathbb{Z}_n^+$ tale che n non soddisfi l'equazione, allora n è certamente composto. Sorprendentemente, il viceversa è quasi valido, così che questo criterio definisce una verifica di primalità quasi perfetta. Si verifica se n soddisfa l'equazione a^{n-1} $(mod\ n)$ per $a = 2$. Se è falsa, si dichiara che n è composto. Altrimenti, *si spera* che n sia primo (anche se, in effetti, si sa che n è o primo o uno pseudoprimo di base 2).

La seguente procedura simula in questo modo il controllo della primalità di n. Usa la procedura MODULAR-EXPONENTIATION del paragrafo precedente. Si assume che l'input n sia un intero maggiore di 2

```
PSEUDOPRIME(n)
  if MODULAR-EXPONENTIATION(2, n-1, n) ≠ 1 (mod n)
    then return COMPOSTO   //Di sicuro.
    else return PRIMO   //Si spera!
```

Questa procedura può produrre degli errori, ma solo di un tipo. Cioé se dice che n è composto, allora è sempre corretta. Se dice che n è primo, allora fa un errore solo se n è uno pseudoprimo di base 2.

Quando questa procedura dà risultati sbagliati? Raramente. Vi sono solo 22 valori di n minori di 10000 per cui sbaglia; i primi quattro di tali valori sono 341, 561, 645 e 1105. Si può mostrare che la probabilità che questo programma faccia un errore su un numero scelto a caso di β bit, tende a zero per $\beta \to \infty$. Usando una stima più precisa dovuta a Pomerance del numero di pseudoprimi di base 2 di una data dimensione, si può stimare che un numero di 50 cifre scelto a caso che è dichiarato primo dalla procedura ha meno di una possibilità su un milione di essere uno pseudoprimo di base 2, e un numero di 100 cifre scelto a caso che è dichiarato primo ha meno di una possibilità su 10^{13} di essere uno pseudoprimo di base 2.

Sfortunatamente, non si possono eliminare tutti gli errori controllando semplicemente l'equazione a^{n-1} $(mod\ n)$ per un secondo numero base, come $a = 3$, perché vi sono interi composti che soddisfano l'equazione per tutte le $a \in \mathbb{Z}_n^*$. Questi interi sono conosciuti come numeri di

Carmichael. I primi tre numeri di Carmichael sono 561, 1105 e 1729. I numeri di Carmichael sono estremamente rari; per esempio, quelli più piccoli di 100000000 sono solo 255.

3.3.3 Verifica di primalità randomizzata di Miller-Rabin

La verifica di primalità di Miller-Rabin supera il problema della semplice verifica PSEUDO-PRIME con due modifiche:

- Prova alcuni valori a di base scelti in modo casuale invece di un solo valore di base.
- Mentre calcola ogni elevamento a potenza modulare, rileva se ha scoperto una radice quadrata non banale di 1, modulo n. Se è così , si ferma e restituisce COMPOSTO.

Lo pseudocodice per il controllo di primalità di Miller-Rabin viene dato di seguito. L' input $n > 2$ è il numero dispari su cui fare la verifica di primalità, ed s è il numero di valori di base da provare, scelti casualmente da $\mathbb{Z}_n^+$. Il codice usa un generatore di numeri casuali. Il codice usa una procedura ausiliaria Witness tale che WITNESS(a, n) è TRUE se e solo se a è un "testimone" del fatto che n è composto, cioé è possibile usare a per dimostrare (come si vedrà) che n è composto. Il controllo WITNESS(a, n) è simile, ma più efficace, del controllo $a^{n-1} \not\equiv 1$ $(mod\ n)$ che è stato usato come base per PSEUDOPRIME (con $a = 2$). Si presenta e si giustifica dapprima la costruzione di WITNESS, e quindi si mostra come la procedura viene usata nella verifica di primalità di Miller-Rabin.

```
WITNESS(a,n)
  sia (b_k, b_{k-1}, ..., b_0) la rappresentazione binaria di n-1
  d ← 1
  for i ← k downto 0
    do x ← d
       d ← (d·d) mod n
       if d = 1 and x ≠ i and x ≠ n-1
          then return TRUE
       if b_i = 1
          then d ← (d·a) mod n
  if d ≠ 1
    then return TRUE
  return FALSE
```

Questo pseudocodice per WITNESS è basato sullo pseudocodice della procedura MODULAR-EXPONENTIATION. La linea 1 determina la rappresentazione binaria di $n-1$, che sarà usata per elevare a alla $(n-1)$ esima potenza. Le linee 3-9 calcolano d come a^{n-1} $(mod\ n)$. Il metodo usato è identico a quello impiegato da MODULAR-EXPONENTIATION. Ogni volta che nella linea 5 è eseguito l'elevamento al quadrato, si controlla nelle linee 6-7 se è stata appena trovata una radice quadrata non banale di 1. Se è così , l'algoritmo si ferma e restituisce TRUE. Le linee 10-11 restituiscono TRUE se il valore calcolato per a^{n-1} $(mod\ n)$ non è uguale a 1, proprio come la procedura PSEUDOPRIME in questo caso restituisce COMPOSTO.

Si deduce ora che se WITNESS(a, n) restituisce TRUE, allora, usando a, può essere costruita una prova che n è composto. Se WITNESS restituisce TRUE dalla linea 11, allora ha scoperto che $d = a^{n-1}$ $(mod\ n) \neq 1$. Se n è primo, però, si ha dal teorema di Fermat che $a^{n-1} \equiv$ $(mod\ n)$ per tutte le $a \in \mathbb{Z}_n^+$. Quindi n non può essere primo, e l'equazione a^{n-1} mod $1 \neq 1$ è una prova

di questo fatto.
Se WITNESS restituisce TRUE dalla linea 7, allora ha scoperto che x è una radice quadrata non banale di 1, modulo n, poiché si ha che $x \not\equiv 1 \ (mod\ n)$ e anche $x^2 \equiv (mod\ n)$. Solo se n è composto ci può essere una radice quadrata non banale di 1, modulo n, così che la dimostrazione che x è una radice quadrata non banale di i modulo n è una prova che n è composto.
Con ciò la dimostrazione della correttezza di WITNESS è completa. Se la chiamata di WITNESS(a, b) dà come risultato TRUE, allora n è sicuramente composto, e una dimostrazione che n è composto può essere facilmente determinata usando a ed n. Si esamina ora la verifica di primalità di Miller-Rabin basata sull' uso di WITNESS

```
MILLER-RABIN(n,s)
  for j ← 1 to s
    do a ← random(1, n-1)
      if WITNESS(a,n)
        then return COMPOSTO   // Di sicuro
  return PRIMO   // Quasi sicuramente
```

La procedura MILLER-RABIN è una ricerca probabilistica di una dimostrazione che n sia composto. Il ciclo principale (a cominciare dalla linea 1) prende s valori casuali di a da $\mathbb{Z}_n^+$ (linea 2). Se una delle a che sono state scelte è un testimone che n è composto, allora MILLER-RABIN restituisce COMPOSTO nella linea 4. Questo risultato è sempre corretto, per la correttezza di WITNESS. Se nessun testimone può essere trovato in s prove, MILLER-RABIN assume che ciò accade perché non vi sono testimoni da trovare, e quindi n è primo. Si vedrà che questo risultato è probabile che sia corretto se s è sufficientemente grande, ma vi è una piccola probabilità che la procedura possa essere sfortunata nella sua scelta delle a e che qualche testimone esista anche se non ne è stato trovato alcuno.

Per mostrare il funzionamento di MILLER-RABIN, sia n il numero di Carmichael 561. Supponendo che $a = 7$ sia scelto come base, la tabella 1 mostra che WITNESS scopre una radice quadrata non banale di 1 nell'ultimo passo di elevamento al quadrato, poiché $a^{280} \equiv 67 \ (mod\ n)$ e $a^{560} \equiv 1 \ (mod\ n)$. Allora, $a = 7$ è un testimone del fatto che n è composto, WITNESS(7, n) restituisce TRUE, e MILLER-RABIN restituisce COMPOSTO. Se n è un numero di β bit, MILLER-RABIN richiede $O(s\beta)$ operazioni aritmetiche e $O(s\beta^3)$ operazioni sui bit, poiché asintoticamente non richiede più lavoro dell'elevamento a potenza modulare.

3.3.4 Tasso di errore del test di primalità di Miller-Rabin

Se MILLER-RABIN restituisce PRIMO, allora vi è una piccola probabilità che abbia commesso un errore. Diversamente da PSEUDOPRIME, però, la possibilità di errore non dipende da n; non vi sono input "cattivi" per questo algoritmo, piuttosto, dipende da quanto è grande s e dalla fortuna nella scelta dei valori di base a. Inoltre, poiché ogni controllo è più stringente di un semplice controllo dell'equazione $a^{n-1} \equiv 1 \ (mod\ n)$, ci si può aspettare che il tasso di errore dovrebbe essere piccolo per interi n scelti a caso. Il seguente teorema presenta un' argomentazione più precisa.

Teorema 3.3 *Se n è un numero dispari composto, allora il numero di testimoni del fatto che n è composto è almeno (n-1)/2.*

Teorema 3.4 *Per un qualunque intero dispari $n > 2$ e un intero s positivo, la probabilità che MILLER-RABIN(n, s) dia un errore è al più 2^{-s}.*

Pertanto, la scelta di $s = 50$ dovrebbe essere sufficiente quasi per qualsiasi applicazione immaginabile. Se si sta cercando di trovare numeri primi grandi applicando la MILLER-RABIN a interi grandi scelti in modo casuale, allora si può dedurre (sebbene non si voglia farlo qui) che scegliendo un valore piccolo di s (come 3) è molto improbabile che porti a risultati erronei. Cioé, per un intero n dispari composto scelto casualmente, il numero medio di non testimoni del fatto che n è composto è probabile che sia molto più piccolo di $(n-1)/2$. Se l'intero n non è scelto in modo casuale però, il meglio che si possa dimostrare è che il numero di non testimoni è al più $(n-1)/4$, usando una versione migliorata del teorema precedente. Inoltre, esistono interi n per cui il numero di non testimoni è $(n-1)/4$.

4 Generazione di chiavi

L'algoritmo di generazione delle chiavi permette di generare la chiave pubblica (e, n) e la chiave privata (d, n). Tale algoritmo può essere schematizzato in 5 punti:

1. Input L (lunghezza modulo)
2. Genera 2 primi di lunghezza L/2
3. $n \leftarrow p \cdot q$
4. scegli $1 < e < \Phi(n)$, comunque piccolo
5. **if** $gcd(e, (p-1)(q-1)) = 1$
   then $d \leftarrow e^{-1}\ mod\ (p-1)(q-1)$
   else goto 4

La lunghezza L definita nel punto 1 indica da quanti bit[5] sarà composto n, è quindi fondamentale scegliere un valore appropriato per garantire una buona sicurezza. I due numeri generati al passo 2 sono di lunghezza L/2 perché il prodotto di tali numeri è di lunghezza L. Nei passi successivi (3,4,5) vengono generati e, d ed n; (e, n) rappresenta la chiave pubblica, (d, n) quella privata.

5 Sicurezza di RSA

La sicurezza della RSA si basa sul fatto che la funzione di cifratura $x^e\ mod\ n$ è una funzione trabocchetto che è computazionalmente difficile da invertire per un nemico che volesse decifrare un messaggio. Solo conoscendo la fattorizzazione di n è possibile trovare il valore delle chiavi. Infatti conoscendo la fattorizzazione di n è possibile calcolare $\Phi(n) = (p-1)(q-1)$ e calcolare $d = e^{-1}\ mod\ \Phi(n)$ usando l'algoritmo esteso di Euclide. Da ciò si deduce che la sicurezza della RSA dipende dal problema di fattorizzare grandi numeri. Di conseguenza il modo più ovvio di attaccare il sistema, è quello di tentare di risolvere quest'ultimo problema. Gli algoritmi più efficienti, conosciuti fino ad ora, hanno una complessità

$$O(e^{\sqrt{(\ln x)(\ln \ln x)}})$$

[5] Bisogna tener conto che spesso si è costretti a lavorare con byte e non bit, visto che le funzioni di lettura da file usano come "unità di misura" i byte.

Fattorizzare un numero di 664 bit richiede almeno 10^{23} passi usando gli algoritmi più efficienti; per cui ipotizzando di avere una rete costituita da un milione di computer che a testa eseguono un milione di passi al secondo, il tempo impiegato per fattorizzare n sarebbe dell'ordine dei 4000 anni. Se poi n fosse un numero a 1024 bit la stessa rete impiegherebbe 10^{10} anni per fattorizzarlo. Ciò sottolinea l'osservazione fatta nell'introduzione e cioè che non è impossibile decifrare un testo cifrato con il crittosistema RSA, ma piuttosto è computazionalmente difficile.

Fattorizzare un numero n significa trovare i suoi fattori primi. Il problema della fattorizzazione è uno dei più vecchi nella teoria dei numeri.

5.1 Attacchi RSA

Osserviamo che per un crittoanalista rompere la RSA equivale a calcolare $\Phi(n)$. Infatti, se n e $\Phi(n)$ sono conosciuti ed n è il prodotto di due primi p e q, n può essere facilmente fattorizzato risolvendo le due seguenti equazioni: $n = p \cdot q$ e $\Phi(n) = (p-1)(q-1)$. Nelle due incognite p e q. Sostituendo $q = n/p$ nella seconda equazione se ne ottiene un'unica di secondo grado nella sola incognita p:

$$p^2 - (n - \Phi(n) + 1)p + n = 0$$

Le due radici di questa equazione sono i fattori p e q. Quindi se un crittoanalista conosce il valore di $\Phi(n)$ può fattorizzare n e rompere il sistema.

Esempio: supponiamo che il crittoanalista conosca $\Phi(n) = 84754668$ e $n = 84773093$. Queste informazioni gli permettono di scrivere l'equazione: $p^2 - 18426p + 84773093 = 0$. Risolvendo l'equazione si ottengono le due radici 9539 e 8887 che rappresentano i fattori p e q di n.

6 Storia della crittografia

L'esigenza di garantire la riservatezza dell'informazione non è un aspetto recente nella vita dell'uomo ma ha caratterizzato la sua storia fin dai tempi greci. I segreti fanno parte della vita umana, così come nascondere le informazioni ed i messaggi in modo che solo persone autorizzate possano accedervi.

Le prime notizie di un sistema di crittografia si possono trovare negli scritti di Plutarco che descrive un metodo di scrittura nascosta adottato al tempo di Lisandro (400 a.C.) dagli spartani. Il metodo consisteva nell'arrotolare una striscia di cuoio attorno ad un bastone detto scitala e caratterizzato da un certo diametro; il messaggio veniva quindi scritto sul cuoio per colonne dall'alto verso il basso, in modo che, una volta srotolata la striscia di cuoio, risultasse un insieme incomprensibile di lettere mischiate. Per leggere il messaggio il destinatario doveva disporre di un bastone dello stesso diametro attorno al quale arrotolava nuovamente la striscia di cuoio per riottenere il messaggio inviato.

Esempio (con cilindretto di altezza pari a 5 caratteri e circonferenza pari a 9).
Testo in chiaro: ALLE PROSSIME ELEZIONI MI PRESENTO ANCHE IO
Cifratura:

A	*P*	*I*	*L*	*N*		*E*	*A*	*I*
L	*R*	*M*	*E*	*I*	*P*	*N*	*C*	*O*
L	*O*	*E*	*Z*		*R*	*T*	*H*	
E	*S*		*I*	*M*	*E*	*O*	*E*	
	S	*E*	*O*	*I*	*S*			

Testo cifrato: APILN EAILRMEIPNCOLOEZ RTH ES IMEOE SEOIS

Ancora nell'antica Grecia, veniva usato il disco di Enea il Tattico, un particolare disco con 24 fori, uno per ogni lettera dell'alfabeto. La codifica del messaggio avveniva passando un filo attraverso i fori corrispondenti alle differenti lettere. Srotolando il filo, avveniva la lettura al contrario del messaggio.

Tuttavia il primo esempio di cifratura vero e proprio è squisitamente italiano, e risale a Giulio Cesare, il quale usava un sistema per comunicare con i propri eserciti in modo da ridurre il rischio di possibili intercezioni non autorizzate. Il suo metodo, geniale per quei tempi ed assolutamente banale con l'attuale tecnologia, consisteva nello scrivere i messaggi spostando tutte le lettere dell'alfabeto di 3 posizioni: A con D, B con E, ... , Z con C. Tale sistema è noto come cifrario monoalfabetico. Per ridare al messaggio la forma originaria occorreva dunque risostituire di nuovo ogni carattere, questa volta con quello che lo precede di tre posizioni nell'ordinamento alfabetico.

Esempio (con alfabeto a 21 caratteri).
Messaggio in chiaro: `DE BELLO GALLICO`
Messaggio cifrato: `GH EHOOR LDOONFR`

Augusto introdusse un sistema di cifratura più complesso di quello di Cesare, utilizzava per scrivere un messaggio il testo scritto in un libro, per esempio l'Odissea. Il sistema consisteva nel cambiare sistema di cifratura per ogni lettera del messaggio, con la convenzione che all'i-esimo carattere da decodificare faceva corrispondere l'i-esimo carattere del libro che usava. Tale sistema usava un cifrario cosiddetto polialfabetico, in cui per ogni carattere si utilizza un cifrario monoalfabetico.

Per tutto il medioevo ed i secoli a seguire, si utilizzarono sistemi sempre più complessi, ma la grande novità arrivò verso la fine dell'Ottocento con la comparsa di macchine in grado di cifrare e decifare automaticamente i messaggi.

La prima macchina di cui si ha notizia è il disco di Wheatstone, un sistema formato da due dischi concentrici, il più esterno su cui punta la lettera in chiaro, ed il più interno con la corrispondente lettera cifrata.

Durante la seconda guerra mondiale i tedeschi utilizzavano una macchina di cifratura chiamata Enigma, composta di vari dischi che ruotavano alla pressione di ogni tasto. Nonostante i tedeschi la considerassero inattaccabile, gli inglesi costruirono una macchina chiamata Colossus in grado di decifrarne i messaggi, contribuendo sicuramente alla sconfitta dei tedeschi nel conflitto.

7 Terminologia crittografica

Il campo della crittografia, solo di recente balzato all'attenzione del grosso pubblico e degli organi di informazione, soffre purtroppo dell'uso di termini errati o imprecisi, spesso modellati erroneamente dai tecnici sui corrispondenti termini inglesi. Ma la crittografia è una disciplina che affonda le sue radici storiche in Italia (gli ambasciatori della Repubblica Veneziana e i Messi Pontifici erano i principali utilizzatori di tecniche crittografiche del Rinascimento...) e dunque dispone di un nutrito ancorché, corretto insieme di termini che vale la pena di usare quando opportuno. E dunque bando agli anglicismi "crittare" o peggio ancora "encrittare", modellati sull'inglese "to encrypt": il verbo corretto è "cifrare" o "mettere in cifra"; mentre l'azione del cifrare si chiama "cifratura", non "crittazione" o "crittografazione". Il sistema di crittografia si chiama semplicemente "cifrario", mentre il messaggio risultante dalla cifratura è un "testo cifrato", in contrapposizione al "testo in chiaro" che è l'originale. La "chiave" è invece la particolare parola, o frase, utilizzata per applicare ad un testo chiaro le regole del cifrario per produrre un testo cifrato. L'operazione inversa alla cifratura si chiama "decifratura" e, attenzione, non è la stessa cosa della "decrittazione"! Fra "decifrare" e "decrittare" vi è infatti una differenza sostanziale: chi decifra è infatti colui il quale è legittimamente in possesso della chiave del cifrario, e la applica secondo le corrette regole per mettere in chiaro un testo cifrato a lui regolarmente destinato; chi decritta, invece, è un estraneo che non è affatto in possesso della chiave, ma tenta fraudolentemente di venire a conoscenza del contenuto di un testo cifrato a lui non destinato utilizzando sistemi indiretti.

Part II
Sorgenti

8 Classe: comuni

Questa classe contiene le funzioni comuni ai tre programmi

8.1 comuni.H

21 ⟨*comuni.H* 21⟩≡

```
//#include "intero.H"
#include <fstream.h>
typedef long int intero;

class comuni
{
public:
  comuni();
  ~comuni();

  int cifre_binario(intero);
  intero potenza_modulare(intero, intero, intero);
  void apri_file(  char*, ifstream& );
  void crea_file( char*, ofstream& );
  double resto(double, double);
};
```

8.2 comuni.cpp

22 ⟨*comuni.cpp* 22⟩≡ 23▷

```
#include "comuni.H"
#include <stdlib.h>
#include <math.h>
#include <string>
#include <stdio.h>

comuni::comuni()
{
}

comuni::~comuni()
{
}

//calcolo il numero di cifre che compongono il numero (in binario)
int comuni::cifre_binario(intero x)
{
  //  cout << endl << "numero : " << x;
  int count = 0;
  while (x != 0)
  {
    x = x >> 1;
    count++;
  }
  return count;
}
```

Calcola la potenza modulare di $a^b \ mod \ n$ utilizzando il metodo delle quadrature ripetute

23 $\langle$*comuni.cpp* 22$\rangle$+≡ ◃22 24▹

```
intero comuni::potenza_modulare(intero a, intero b, intero n)
{
  intero mask, count = 0;
  double d = 1;
  intero c = 0;

  count = cifre_binario(b);
  mask = (intero)pow(2, count-1);

  for (int i=count; i>0; i--)
  {
    c = 2*c;
    d = resto( (d*d), n);
    //cerr << d << " ";
    if (b&mask)
    {
      c++;
      d = resto( (d*a), n);
    }
    mask = mask >> 1;
  }
  //cout << "\na: " << a;
  return (intero)d;

}
```

Crea un file dove salvare i dati elaborati. La stringa `str` contiene la parte finale del messaggio da stampare.

24 ⟨*comuni.cpp* 22⟩+≡ ◁23 25▷

```
void comuni::crea_file( char * str, ofstream& ofs )
{
  char nome[256];

  for(;;)
  {
    cout << "\n Dammi il nome del file" << str;
    cin >> nome;

    ofs.open( nome, ios::noreplace|ios::binary|ios::out );

    if ( !ofs )  // se il file esiste gi
    {
      cerr << "\n Accesso negato ( " << nome << " ) ";
      perror("");

      cout << " vuoi sovvrasvriverlo [s/n]: ";
      char valido[10];
      cin >> valido;

      if ( 0 == strncmp("s", valido, 1 ) )
      {
        ofs.close();
        ofs.clear();
        ofs.open( nome, ios::binary|ios::out );

        if ( !ofs ) // se file di sola lettra o altro
        {
          cerr << "\n Accesso negato ( " << nome << " ) ";
          perror("");
          ofs.close();
          ofs.clear();
          continue;
        }
        return;

      }
      else  // non voglio sovrascrivere
      {
        ofs.close();
        ofs.clear();
        continue;
      }
    }
```

```
    return;
  }
}
```

Apre un file da dove estrarre i dati. La stringa `str` contiene la parte finale del messaggio da stampare.

25 ⟨*comuni.cpp* 22⟩+≡ ◃24 26▹

```
void comuni::apri_file( char* str, ifstream& ifs)
{
  char nome[256];

  for(;;)
  {
    cout << "\n Dammi il nome del file" << str;
    cin >> nome;
    ifs.open( nome, ios::binary|ios::nocreate|ios::in  );

    if( !ifs )
    {
      cerr << "\n Accesso negato ( " << nome << " ) ";
      perror("");
      ifs.close();
      ifs.clear();
      continue;
    }
    return;
  }
}
```

Calcola il resto della divisione fra due double, se invece di ritornare `r` si ritornasse `q` la funzione calcolerebbe la divisione.

26 ⟨*comuni.cpp* 22⟩+≡ ◃25

```
double comuni::resto(double x, double y)
{
  double r = x;
  double q = 0;
  double w = y;

  while( w<=x ) w = w*10;
  while( w!=y )
  {
    q=q*10;
    w=w/10;
    while( w<=r )
    {
      r = r-w;
      q=q+1;
    }
  }
  return r;
}
```

9 Programma: chiavi

Questo programma si occupa della generazione delle chiavi. Ha due modalità di funzionamento: la prima come un tipico programma UNIX, l'altra interattiva. Passando direttamente il nome delle due chiavi il programma crerà due file contenenti le chiavi. ATTENZIONE se presenti file con lo stesso nome verranno sovrascritti! Questo non accade con la versione interattiva che chiede conferma.

9.1 Primi.H

27a ⟨*Primi.H* 27a⟩≡

```
#include "comuni.H"

class Primi: public comuni
{

public:
  Primi();
  ~Primi();

  intero primo();

private:
  bool witness(intero, intero);
  bool miller_rabin(intero, intero);

};
```

9.2 Primi.cpp

27b ⟨*Primi.cpp* 27b⟩≡ 28 ▷

```
#include <iostream.h>
#include <limits.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "Primi.H"
#include "Rand.H"

Primi::Primi()
{
        //inizializzo il generatore di numeri casuali con l'ora attuale
        smyrand( (unsigned)time( NULL ) );
}

Primi::~Primi(){;}
```

Verifica di primalità con l'algoritmo di Miller-Rabin precedentemente illustrato

28 ⟨*Primi.cpp* 27b⟩+≡ ◁27b

```
bool Primi::miller_rabin(intero n, intero s)
{
  intero j, a;

  for(j = 1; j<=s; j++)
  {
    a=dado(n-1);
    //cout << "\n a: " << a;
    if (witness(a, n)) return false;
  }
  return true;
}

bool Primi::witness(intero a, intero n)
{
  int i, n_cif_bin;
  intero mask;
  intero x=0, rap_bin=0;
  long double d = 1;
  //cerr << "\nwitness\a d=" << d << " x=" << x << " n-1=" << (n-1);
  rap_bin = n-1;
  n_cif_bin = cifre_binario(rap_bin);

  mask = (intero)pow(2, n_cif_bin-1 );

  for (i = n_cif_bin; i>0; i--)
  {
    x = (intero)d;  //e' in modulo n
    d = resto((d*d),n);
    //cerr << "\nnel ciclo\a d=" << d << " x=" << x << " n-1=" << (n-1);
    //cout << d << " ";
    if ((d==1) && (x!=1) && ( x!=(n-1) ) ) return true;   //composto
    //cerr << "\nnel ciclo\a: " << n;
    if (rap_bin&mask) d=resto((d*a),n);
    mask = mask >> 1;
  }

  if (d != 1) return true; //coposto
  return false;   // probabilmente primo
}


intero Primi::primo()
{
  intero tmp = 0;
  //smyrand( (unsigned)time( NULL ) );
```

```
  //cout << LONG_MAX << endl;
  //cout << RAND_MAX << endl;

  while (1)
  {
    tmp = dado(9800); // deve essere minore di sqrt(LONG_MAX)
    // ho dovuto usare un numero minore di 9800 per prob di over flov
    //cout << endl << tmp;
    if (tmp<257) continue; //e' maggiore di sqrt(SHRT_MAX)

    if ((tmp%2)==0) tmp--;  //se era pari

    if (miller_rabin(tmp, 50))  //50 e' il numero cha aumenta l'affidabilita'
    {
      //cerr << "\nnumero primo: " << tmp;
      return tmp;
    }
  }
}
```

9.3 Chiavi.H

30 $\langle$*Chiavi.H* 30$\rangle\equiv$

```
#include "Primi.H"
#include <fstream.h>

/*
class Euclide
{
public:
  intero MCD;
  intero x;
  intero y;
};
*/
class key
{
public:
  intero e;
  intero d;
  intero n;
};

class Chiavi: public comuni
{
public:
  void salva();
  void salva(char*, char*);

private:
  key genera_key();
  intero MCD(intero, intero);
  intero getMaxSteps(intero);
  intero extendedEuclidean (intero, intero);
};
```

9.4 Chiavi.cpp

31 ⟨*Chiavi.cpp* 31⟩≡

```
#include "Chiavi.H"
#include <fstream.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include "Rand.H"

intero Chiavi::MCD(intero fi, intero e)
{
  intero r;
  while(e>0)
  {
    r=(fi%e);
    fi=e;
    e=r;
  }
  return fi;
}

/*
//usa l'algoritmo di esteso di Euclide
Euclide Chiavi::MCD(intero fi, intero e)
{
  Euclide euclide;
  Euclide tmp;

  if (e == 0)
  {
    euclide.MCD = fi;
    euclide.x = 1;
    euclide.y = 0;

    return euclide;
  }

  tmp = MCD( e, (fi%e) );

  euclide.MCD = tmp.MCD;
  euclide.x = tmp.y;
  euclide.y = (tmp.x-( (fi/e)*tmp.y) );
  //cerr << "\nMCD: " << euclide.MCD;
  return euclide;
}
*/
```

```
// number of steps in the Euclidean algorithm is proven to be
// at most 5 times the number of digits in the smaller number
intero Chiavi::getMaxSteps (intero enc)
{
  intero steps = 1;
  while (pow(10,steps) < enc)
    steps++;
  return 5*steps;
}

// Pre: phi > enc
// Post: returns n such that
// GCD(phi,e) + k*phi = n*enc for some positive int k
intero Chiavi::extendedEuclidean(intero phi, intero enc)
{
  int count = 0;
  intero temp = 0;
  intero a = phi;
  intero b = enc;
  intero *coeffic = new intero[getMaxSteps(enc)];

  while (b > 0)
  {
    coeffic[count] = a/b;
    temp = a;
    a = b;
    b = temp%b;
    count++;
  }

  count--;
  intero k = 1;                  // k  in  k*phi
  intero n = -coeffic[--count];  // n  in  n*enc
  intero k_old = 0;

  while (count > 0)
  {
    k_old = k;
    k = n;
    n = k_old - k*coeffic[--count];
  }

  if (n >= 0) return n;
  else return (phi + n);  // essentially -n % phi
}


key Chiavi::genera_key()
```

```
{
  intero n, q, p, fi;
  intero e = 0;
  Primi tmp;
  key key;

  // genero i due numeri primi minori di sqrt(LONG_MAX)
  // necessari per ottenere n
  q = tmp.primo();
  p = tmp.primo();

  n = q*p;
  key.n = n;

  // adesso calcolo la funzione di eulero f(n)
  fi = (p-1)*(q-1);
  //cerr << "\nfi: " << fi;

  // mi serve un numero dispari e "piccolo" primo con fi

  for(;;)
  {
    e = dado(128);

    if ( e<=2 ) continue;

    if ( (e%2)==0 ) e++;  //se era pari
    //cerr << "\ne: " << e;
    //euclide = MCD(e, fi);

    if ( MCD(e, fi)==1 )
    {
      key.e = e;
      key.d = extendedEuclidean(fi, e);
      //cerr << "\nx: ";
      if ( 1 == (key.e*key.d)%fi ) // se d e' il reciproco di e
      {
        // test di corettezza !
        intero C = potenza_modulare(32000, key.e, key.n);
        intero D = potenza_modulare(65000, key.e, key.n);
        if ( (32000 != potenza_modulare(C, key.d, key.n)) || (65000 != potenza_modulare(D, key.d, key.n)) )
        {
          cerr << "\a\n Probabie errore nella fase di creazione delle chiavi!\n";
      //    cerr << C << " " << D << endl;
        }
        cout << "\n\n chiave pubblica\n n: " << key.n << "\n e: " << key.e;
        cout << "\n\n chiave privata \n n: " << key.n << "\n d: " << key.d << endl << endl;
        return key;
      }
```

```
    }
  }
}


// se ho passato i nomi da riga di comando
void Chiavi::salva( char* key_pub, char* key_priv )
{
  key key;
  key = genera_key();

  ofstream ofs_pub( key_pub, ios::binary|ios::out );
  if ( !ofs_pub )
  {
    cerr << "\n Accesso negato ( " << key_pub << " ) ";
    perror("");
    ofs_pub.close();

    cerr << "\nCreazione non effetuata !\n";
    return;
  }

  ofs_pub << key.n << endl << key.e << endl;
  ofs_pub.close();

  ofstream ofs_priv( key_priv, ios::binary|ios::out );
  if ( !ofs_priv )
  {
    cerr << "\n Accesso negato ( " << key_priv << " ) ";
    perror("");
    ofs_priv.close();
    cerr << "\nCreazione non effetuata !\n";
    return;
  }
  ofs_priv << key.n << endl << key.d << endl;
  ofs_priv.close();

  return;
}


//se i nomi non sono sati indicati da riga di comando
void Chiavi::salva()
{
  cout << "\n\a";
  cout << " ****************************************************************\n";
  cout << " *                                                              *\n";
  cout << " *                       Crittografia RSA                       *\n";
  cout << " *       Progetto realizzato da: Edoardo  Di Lorenzo 1SI        *\n";
```

```
  cout << " *                        per l'esame di                        *\n";
  cout << " *   laboratorio di informatica: algoritmi e strutture di dati  *\n";
  cout << " *                        A.A. 2000/2001                        *\n";
  cout << " *           Modulo per la creazione delle chiavi               *\n";
  cout << " *                                                              *\n";
  cout << " ****************************************************************\n";

  key key;
  key = genera_key();

  ofstream ofs;
  crea_file(" per la chiave pubblica: ", ofs);
  ofs  << key.n << endl << key.e << endl;
  ofs.close();
  ofs.clear();

  crea_file(" per la chiave privata: ", ofs);
  ofs << key.n << endl << key.d << endl;
  ofs.close();
}
```

9.5 genkey.cpp

35 ⟨*genkey.cpp* 35⟩≡

```
#include "Chiavi.H"
#include <iostream.h>

void main(int argc, char** argv )
{
  Chiavi Key;

  switch ( argc )
  {
  case 3: Key.salva( argv[1], argv[2] );
    break;
  case 1: Key.salva();
    break;
  default:
    {
      cerr << "\n Numero di parametri errato";
      cout << "\n Usa: chiavi chiave_publica chiave_privata\n\n";
    }
  }
}
```

10 Programma: cifra

Questo programma si occupa della cifratura di un file. Anche cifra ha due modalità di funzionamento come chiavi. Questo programma richiede tre argomenti: il file da cifrare, il nome del file di output e quello in cui è contenuta la chiave. Cifra è in grado di cifrare tutti i tipi di file sia binari che in codifica ASCII. Se il file è di grosse dimensioni l'operazione potrebbe richiedere un po' di tempo.

10.1 Codifica.H

36 ⟨*Codifica.H* 36⟩≡

```
#include "comuni.H"

class Codifica: public comuni
{
public:
  void cifra();
  void cifra(char *, char *, char *);
};
```

10.2 Codifica.cpp

37 ⟨*Codifica.cpp* 37⟩≡

```
#include "Codifica.H"
#include <stdlib.h>
#include <stdio.h>

// versione interattiva
void Codifica::cifra()
{
  intero n = 0;
  intero e = 0;
  unsigned short buff;
  intero tmp;

  cout << "\n\a";
  cout << " *****************************************************************\n";
  cout << " *                                                               *\n";
  cout << " *                       Crittografia RSA                        *\n";
  cout << " *       Progetto realizzato da: Edoardo  Di Lorenzo 1SI        *\n";
  cout << " *                         per l'esame di                        *\n";
  cout << " *   laboratorio di informatica: algoritmi e strutture di dati  *\n";
  cout << " *                         A.A. 2000/2001                        *\n";
  cout << " *              Modulo per la cifrazione di un file              *\n";
  cout << " *                                                               *\n";
  cout << " *****************************************************************\n";

  ifstream ifs;

  // estraggo la chiave
  apri_file( " in cui e' contenuta la chiave: ", ifs);
  ifs >> n >> e;
  ifs.close();
  ifs.clear();

  //cerr << "\nn: " << n << "    e: " << e;

  apri_file( " da cifrare: ", ifs);

  ofstream ofs;
  crea_file(" di output: ", ofs);

  cerr << "\n Inizio fase di cifratura!\n";

  while(!ifs.eof() )
  {
  ifs.read( (char *) &buff, sizeof(buff) );
  //cerr << buff << " " << potenza_modulare(buff, e, n) << endl;
  tmp = potenza_modulare(buff, e, n);
```

```
  ofs.write( (char *) &tmp, sizeof(tmp) );
  }

  ifs.close();
  ofs.close();

  cout << "\n Cifratura terminata! \a\n\n";
}

//versione "stile" programma UNIX
void Codifica::cifra( char* input, char* output, char* chiave)
{
  unsigned short buff;
  intero tmp;
  intero n = 0;
  intero e = 0;

  ifstream ifs( input, ios::binary|ios::nocreate|ios::in );
  if( !ifs )
  {
    cerr << "\n Accesso negato ( " << input << " ) ";
    perror("");
    ifs.close();
    cerr << "\n Operazione non effetuata !\n";
    return;
  }

  ofstream ofs( output, ios::binary|ios::out );
  if ( !ofs )
  {
    cerr << "\n Accesso negato ( " << output << " ) ";
    perror("");
    ofs.close();

    cerr << "\nOperazione non effetuata !\n";
    return;
  }

  ifstream ifs_chiave( chiave, ios::binary|ios::nocreate|ios::in );
  if( !ifs )
  {
    cerr << "\n Accesso negato ( " << chiave << " ) ";
    perror("");
    ifs_chiave.close();
    cerr << "\n Operazione non effetuata !\n";
    return;
  }

  cout << "\n Inizio fase di cifratura!\n";
```

```
  ifs_chiave >> n >> e;
  ifs_chiave.close();

  while(!ifs.eof() )
  {
  ifs.read( (char *) &buff, sizeof(buff) );
  //cerr << buff << " " << potenza_modulare(buff, e, n) << endl;
  tmp = potenza_modulare(buff, e, n);
  ofs.write( (char *) &tmp, sizeof(tmp) );
  }

  ifs.close();
  ofs.close();

  cout << "\n Cifratura terminata! \a\n\n";
}
```

10.3 cifrafile.cpp

39 ⟨*cifrafile.cpp* 39⟩≡

```
#include "Codifica.H"
#include <iostream.h>

void main(int argc, char** argv)
{
  Codifica cifra;
  switch ( argc )
  {
  case 4: cifra.cifra( argv[1], argv[2], argv[3] );
    break;
  case 1: cifra.cifra();
    break;
  default:
    {
      cerr << "\n Numero di parametri errato";
      cout << "\n Usa: cifra file_da_cifrare file_output chiave\n\n";
    }
  }
}
```

11 Programma: decifra

Questo programma si occupa della decifratura di un file. Come i precedenti ha due modalità di funzionamento. Per funzionare ha bisogno di tre argomenti: il file da decifrare, il nome del file di output, contenete il testo (o altro) in chiaro, e quello della chiave.

11.1 decifra.H

40 ⟨*decifra.H* 40⟩≡

```
#include "comuni.H"

class decifra : public comuni
{
public:
  decifra();
  ~decifra();
  void go();
  void go( char*, char*, char* );
};
```

11.2 decifra.cpp

41 ⟨*decifra.cpp* 41⟩≡

```
#include "decifra.H"
#include <stdlib.h>
#include <stdio.h>

decifra::decifra()
{

}

decifra::~decifra()
{

}

//se non ho passato argomenti: versione interattiva
void decifra::go()
{
  intero n = 0;
  intero d = 0;
  intero buffi;
  unsigned short tmp;

  cout << "\n\a";
  cout << " ****************************************************************\n";
  cout << " *                                                              *\n";
  cout << " *                      Crittografia RSA                        *\n";
  cout << " *       Progetto realizzato da: Edoardo  Di Lorenzo 1SI        *\n";
  cout << " *                        per l'esame di                        *\n";
  cout << " *   laboratorio di informatica: algoritmi e strutture di dati  *\n";
  cout << " *                        A.A. 2000/2001                        *\n";
  cout << " *            Modulo per la decifrazione di un file             *\n";
  cout << " *                                                              *\n";
  cout << " ****************************************************************\n";

  ifstream ifs;

  // estraggo la chiave
  apri_file( " in cui e' contenuta la chiave: ", ifs);
  ifs >> n >> d;
  ifs.close();
  ifs.clear();

  apri_file( " da decifrare: ", ifs);

  ofstream ofs;
  crea_file(" di output: ", ofs);
```

```
  cerr << "\n Inizio fase di decifratura!\n";

  while( !ifs.eof() )
  {
    ifs.read( (char *) &buffi, sizeof(buffi) );
    if ( EOF == ifs.peek() ) continue;
    //cerr << buffi << "\t" << (char)potenza_modulare(buffi, d, n)<< endl;
    tmp = (unsigned short)potenza_modulare(buffi, d, n);
    ofs.write( (char*) &tmp, sizeof(tmp) );
  }

  ifs.close();
  ofs.close();

  cout << "\n Decifratura terminata! \a\n\n";
}

// ho passato le informazioni necessarie a riga di comando
void decifra::go(char* input, char* output, char* chiave)
{
  intero n = 0;
  intero d = 0;
  intero buffi;
  unsigned short tmp;

  ifstream ifs( input, ios::binary|ios::nocreate|ios::in );
  if( !ifs )
  {
    cerr << "\n Accesso negato ( " << input << " ) ";
    perror("");
    ifs.close();
    cerr << "\n Operazione non effetuata !\n";
    return;
  }

  ofstream ofs( output, ios::binary|ios::out );
  if ( !ofs )
  {
    cerr << "\n Accesso negato ( " << output << " ) ";
    perror("");
    ofs.close();

    cerr << "\nOperazione non effetuata !\n";
    return;
  }

  ifstream ifs_chiave( chiave, ios::binary|ios::nocreate|ios::in );
  if( !ifs )
```

```
  {
    cerr << "\n Accesso negato ( " << chiave << " ) ";
    perror("");
    ifs_chiave.close();
    cerr << "\n Operazione non effetuata !\n";
    return;
  }
  cerr << "\n Inizio fase di decifratura!\n";

  ifs_chiave >> n >> d;
  ifs_chiave.close();

  while( !ifs.eof() )
  {
    ifs.read( (char *) &buffi, sizeof(buffi) );
    if ( EOF == ifs.peek() ) continue;
    //cerr << buffi << "\t" << (char)potenza_modulare(buffi, d, n)<< endl;
    tmp = (unsigned short)potenza_modulare(buffi, d, n);
    ofs.write( (char*) &tmp, sizeof(tmp) );
  }

  ifs.close();
  ofs.close();

  cout << "\n Decifratura terminata! \a\n\n";
}
```

11.3 decodifica.cpp

43 ⟨*decodifica.cpp* 43⟩≡

```
#include "decifra.H"

void main(int argc, char** argv)
{
  decifra decifra;
  switch ( argc )
  {
  case 4: decifra.go( argv[1], argv[2], argv[3] );
    break;
  case 1: decifra.go();
    break;
  default:
    {
      cerr << "\n Numero di parametri errato ";
      cout << "\n Usa: decifra file_da_decifrare file_output chiave \n\n";
    }
  }
}
```

References

[1] Domenico Luminati – *Diario*. Appunti delle lezioni di Matematica discreta 2, A.A. 1999-2000

[2] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L., – *Introduction to Algorithms* (cap. 33), MIT Press, Cambridge, MA; McGraw-Hill Book Co., New York (1990)

[3] Concetti di base sulla crittografia e l'infrastruttura a chiave pubblica. Tratto dalle pagine del sito di Microsoft Windows 2000 Server

[4] Materiale trovato con il motore di ricerca *google*