Alessandro Languasco
Alessandro Zaccagnini

CRITTOGRAFIA

Progetto Nazionale Lauree Scientifiche
Regione Veneto
Sottoprogetto Matematica

A. Languasco - A. Zaccagnini
CRITTOGRAFIA

Terza Edizione corretta; Gennaio 2018

La fotografia sullo sfondo della copertina raffigura parte del Disco Cifrante di Leon Battista Alberti, mentre, in primo piano, è rappresentato il "Cryptographer" di Charles Wheatstone. In quarta di copertina sono riprodotti il crittogramma di Edgar Allan Poe riportato dal libro "Lo scarabeo d'oro" ed il crittogramma di Giulio Verne riportato dal libro "Viaggio al centro della Terra". La traslitterazione in lettere latine di quest'ultimo si trova a pagina 42.

Il Progetto Lauree Scientifiche

Nel 2003 il Ministero dell'Istruzione e dell'Università convocò i Presidi delle Facoltà di Scienze ed i rappresentanti di Confindustria per varare il Progetto Lauree Scientifiche, un'iniziativa promozionale da realizzarsi, a cura delle Università, nelle scuole secondarie italiane, per aumentare l'interesse dei giovani alle scienze "dure": matematica, fisica, chimica, scienza dei materiali. Scopo non trascurabile: aumentare le iscrizioni ai corrispondenti corsi di laurea.

Pur rispettando criteri nazionali comuni, il progetto è stato realizzato per discipline separate e su base regionale, come collaborazione tra le locali Università e l'Ufficio Scolastico Regionale. Nel Veneto, per la matematica, sono stati anzitutto scelti 15 istituti secondari, un paio per provincia. In ciascuno di questi poli si è costituito un gruppo di progetto, composto da tre docenti secondari - particolarmente esperti - e due universitari, professori o giovani ricercatori, afferenti agli Atenei di Padova, Verona e Venezia. In ogni polo si è progettato e realizzato un Laboratorio matematico cosí organizzato: si recluta, su base volontaria, un piccolo gruppo di studenti (15-20) ai quali viene proposto, in orario extrascolastico, un capitolo di matematica non tradizionale che si presta allo studio di un problema concreto. Dopo una breve introduzione dei docenti, i ragazzi si mettono al lavoro, penna e computer, per elaborarne una soluzione. Il lavoro svolto in questi laboratori, un pomeriggio la settimana per circa un mese, può essere poi esposto nelle classi al completo, in orario normale, a cura degli stessi giovani protagonisti.

I singoli problemi sono stati scelti in ciascun polo in modo autonomo, tra quelli proposti dai collaboratori universitari. Il tema crittografia è risultato particolarmente attraente, adottato da quattro poli (Rovigo, Castelfranco, Schio, Treviso). Pur pianificando il lavoro in modo autonomo, i quattro gruppi hanno seguito una stessa linea, messa a punto dal prof. Alessandro Languasco, dell'Università di Padova.

Il presente volumetto descrive questo percorso teorico-didattico. Al lettore non si richiedono particolari prerequisiti: introdotti i concetti fondamentali dell'aritmetica moderna, si perviene rapidamente ai metodi crittografici attuali. La parte finale ripartisce il lavoro didattico in cinque incontri, fornendo cosí una guida pratica a chi volesse imitare l'esperimento didattico.

Alla stesura di questa pubblicazione, accanto al prof. Languasco, ha contribuito un altro specialista di teoria dei numeri e crittografia, il prof. Alessandro Zaccagnini, dell'Università di Parma. Ad entrambi va il ringraziamento della direzione e dei collaboratori del P.L.S. Oltre a rendersi utile nell'ambito del progetto, questo libro può servire da buon esempio e incoraggiamento alla produzione di libri e materiali per l'alta divulgazione, un compito importante e difficile al quale i matematici, soprattutto italiani, si sono troppo spesso sottratti.

Benedetto Scimemi,
coordinatore scientifico per il Veneto del P.L.S. per la Matematica

SOMMARIO

PROGETTO LAUREE SCIENTIFICHE – CRITTOGRAFIA
INTRODUZIONE

Nella prima parte di queste note presentiamo alcuni argomenti teorici necessari allo sviluppo di una tematica per il Progetto Nazionale Lauree Scientifiche 2005-2006. Dopodiché analizzeremo velocemente uno strumento di calcolo utile per svolgere esercitazioni al computer (PARI/Gp). Nella terza parte svilupperemo una proposta di cinque Incontri relativi a tale tematica.

Queste note sono principalmente rivolte al personale docente delle Scuole Superiori e sono state sviluppate per costituire una "base comune" di conoscenza ed uno stimolo a sviluppare altro materiale. È chiaro quindi che gli Insegnanti potranno integrarle o snellirle a seconda delle loro esigenze al fine di meglio adattare la presentazione e le esercitazioni alla preparazione dei loro studenti.

Per quanto riguarda le integrazioni, riteniamo che potrebbero essere

(1) individuati ulteriori esempi ed applicazioni che possano aiutare nella presentazione degli argomenti; ragionevolmente essi dovrebbero essere dipendenti dai programmi scolastici svolti nei singoli Istituti e/o Licei in cui il particolare progetto verrà posto in essere; per tale ragione pensiamo che forse essi possano essere meglio individuati dai colleghi delle Scuole Superiori che conoscono "sul campo" la situazione dei programmi e la preparazione degli studenti;

(2) individuati alcuni piccoli programmi da svolgere con l'ausilio del software presente nei Laboratori dei vari Istituti e/o Licei. A tal fine suggeriamo di utilizzare PARI/Gp per le sue peculiari caratteristiche di software dedicato alla Teoria dei Numeri.

Per quanto riguarda gli snellimenti: essenzialmente essi riguardano le quantità di dimostrazioni da svolgere nelle lezioni. A questo livello pensiamo che fornire un esempio significativo degli argomenti cruciali della dimostrazione abbia forse una valenza didattica più consistente. Abbiamo cercato comunque di inserire il numero maggiore possibile di dimostrazioni cercando di semplificarle ove fosse ragionevole. Chiaramente abbiamo omesso quelle che abbiamo ritenuto troppo difficili, tecniche o non particolarmente illuminanti.

In generale, gli argomenti seguono l'impostazione data nel testo [6], negli articoli [9], [10] e [11] e nella conferenza [13].

In questa versione, per esigenze di stampa, troverete in bianco e nero sia le figure che le casistiche relative all'analisi di frequenza dell'esempio 1.7. Le corrispondenti versioni a colori sono incluse nel testo degli articoli precedentemente menzionati.

Inoltre gli script in PARI/Gp descritti nel Capitolo 3 sono anche disponibili in rete nelle pagine web

`http://www.math.unipd.it/~languasc/intro-crittografia/Software.html`

`http://people.dmi.unipr.it/alessandro.zaccagnini//crittografia/Software.html`

Infine, durante lo sviluppo del primo anno del Progetto Lauree Scientifiche, alcuni membri dei vari gruppi del progetto Crittografia del Veneto hanno redatto altre note collegate a queste. Esse si trovano all'indirizzo web `http://www.math.unipd.it/~languasc/PNLS/`).

Ringraziamenti. Desideriamo ringraziare il Prof. S. Antoniazzi (ITIS "Planck" di Lancenigo, Treviso), la Prof.ssa G. Carnovale e la la Prof.ssa L. Fiorot (entrambe dell'Università di Padova) per aver letto una prima versione di queste note ed avere suggerito modifiche e miglioramenti.

Inoltre ringraziamo il Prof. R. Colpi (Università di Padova) per aver suggerito alcuni miglioramenti, il Prof. A. Centomo (Liceo "Corradini", Thiene) per alcuni suggerimenti su comandi UniX utilizzabili per l'analisi di frequenza ed il Prof. G. Pavarin (Liceo "Paleocapa", Rovigo) per aver individuato un errore in uno script PARI/GP.

0.1 Nuova Edizione del 2014

Il Progetto Lauree Scientifiche ha cambiato nome assumendo quello di "Piano Nazionale Lauree Scientifiche".

A seguito di un rinnovato interesse per le tematiche descritte in questo testo, si è deciso di effettuare un aggiornamento di questo documento.

A parte un piccolo rinnovamento grafico e la correzione di alcuni errori di battitura, il testo dei Capitoli 1 e 3 non è essenzialmente cambiato dalla versione del 2006. Maggiori, ma non sostanziali, modifiche sono state apportate al Capitolo 2 in modo da aggiornarlo alla versione del software attualmente disponibile. La bibliografia è stata anch'essa aggiornata aggiornando i vari link ed inserendo il testo di Singh [12] che presenta vari aspetti storici dello sviluppo della Crittografia.

(Giugno 2014, A. Languasco e A. Zaccagnini)

0.2 Nuova Edizione del 2018

Un nuovo aggiornamento del Capitolo 2 è stato effettuato in modo da renderlo aderente alla versione del software attualmente disponibile. Sono state inoltre inserite le referenze bibliografiche [7, 8] e sono stati corretti alcuni link.

(Gennaio 2018, A. Languasco e A. Zaccagnini)

CAPITOLO 1

DISPENSINE DI CRITTOGRAFIA

In queste note forniremo alcune nozioni di base della *Teoria Elementare dei Numeri* ed applicheremo alcuni di tali concetti alla *Crittografia*. Dopo aver introdotto la *Teoria delle Congruenze*, il *Piccolo Teorema di Fermat* ed il *Teorema di Eulero-Fermat*, analizzeremo da un punto di vista storico la nozione di Crittografia. Esaminate le caratteristiche dei Crittosistemi Classici, passeremo ad evidenziare quali siano le fondamentali proprietà dei sistemi Crittografici Moderni. In particolare ci soffermeremo sulla nozione di *Crittografia a Chiave Pubblica* o *Asimmetrica* e ne esamineremo un esempio significativo basato sul fatto che in $\mathbb{Z}$ gli algoritmi di primalità sono nettamente più "veloci" di quelli di fattorizzazione (sistema RSA).

1.1 IL TEOREMA FONDAMENTALE DELL'ARITMETICA E IL TEOREMA DEI NUMERI PRIMI

Ricordiamo per prima cosa la seguente

Definizione 1.1. *Un intero $p > 1$ si dice* primo *se e solo se p è divisibile solamente per* 1 *e per se stesso.*

Si noti che 1 non è un numero primo. Elenchiamo qui alcune ragioni per cui si suole escludere 1 dall'insieme dei numeri primi: prima di passare al dettaglio, è bene osservare che in Matematica si cerca di dare definizioni utili e generali, anche al costo di darle in modo apparentemente "non naturale." Non è certamente pensabile che i Matematici siano costretti a conservare la definizione vista nelle Scuole Medie, quando questa confligge con il principio generale appena esposto.

Veniamo dunque al nostro problema; il numero 1 non viene considerato primo per vari motivi, fra i quali citiamo quelli che riteniamo più importanti.

1. Il numero 1 ha un solo divisore, mentre tutti i numeri primi ne hanno due. Questo è solo un esempio di un fenomeno generale: per molte funzioni aritmetiche assolutamente naturali, come la funzione φ di Eulero definita dalla cardinalità dell'insieme degli interi $0 \leq a < n$ tali che $(a,n) = 1$, sarebbe necessario avere due formule distinte, una valida per 1 e l'altra per i numeri primi $p \geq 2$. In questo caso, infatti, dovremmo dire che $\varphi(p) = p-1$ per *tutti* i $p \geq 2$, ma $\varphi(1) = 1$.

2. Molti teoremi, per esempio il Teorema Fondamentale dell'Aritmetica 1.1, dovrebbero essere enunciati in un modo molto più complicato per tener conto delle proprietà speciali di 1.

3. Nel crivello di Eratostene, descritto nel §1.2, se il numero 1 fosse considerato primo si cancellerebbero *tutti* i numeri tranne lo stesso 1 al primo passo.

4. Un'altra funzione importante è la funzione Ω, che conta il numero *totale* dei fattori primi di un intero $n \geq 2$: in altre parole, se n ha la fattorizzazione $\prod_{i=1}^{k} p_i^{\alpha_i}$, p_i primi, $0 < p_i < p_{i+1}$, $\alpha_i \in \mathbb{N} \setminus \{0\}$, la funzione $\Omega(n)$ vale $\alpha_1 + \alpha_2 + \cdots + \alpha_k$. Se 1 fosse primo, potremmo includere nella fattorizzazione di n la potenza 1^{α}, con $\alpha \in \mathbb{N}$ arbitrario, e quindi $\Omega(n)$ sarebbe indeterminato.

5. Nell'Algebra, gli elementi invertibili degli anelli (cioè quegli elementi a per i quali si può risolvere l'equazione $ax = 1$) hanno uno *status* speciale. In $\mathbb{N}$ l'unico elemento invertibile è proprio 1; esso va quindi trattato a parte.

In definitiva, possiamo riassumere la nostra argomentazione cosí: se decidessimo di considerare primo anche 1, dovremmo rassegnarci a fare continue eccezioni perfino nelle definizioni o nei teoremi più semplici. Per economia, dunque, preferiamo dare una definizione che a prima vista può sembrare meno naturale, ma con la quale non c'è questa necessità. In effetti si tratta di un principio generale della Matematica: l'utilità e la versatilità delle definizioni sono decidibili solo "a posteriori", cioè solo dopo averle viste all'opera e confrontate con possibili definizioni alternative.

Il concetto di primalità è di rilevanza fondamentale, come può già essere esemplificato dal famoso

Teorema 1.1 (Fondamentale dell'Aritmetica). *Ogni intero $n \geq 2$ ha una decomposizione (unica a meno dell'ordine) in fattori primi.*

Dim. Per prima cosa proviamo per induzione l'esistenza della decomposizione.

- passo 1): 2 è primo;

- passo 2): Sia vero che ogni $m < n$ è un prodotto di primi. Dimostriamo che n è prodotto di primi. Se n è primo allora la tesi induttiva è verificata. Sia allora $n = ab$ con $a, b < n$ interi. Ma sia a che b sono prodotto di primi per ipotesi induttiva; quindi anche $n = ab$ è prodotto di primi e la tesi induttiva è verificata. Quindi il procedimento di induzione ci consente di affermare che ogni $n \in \mathbb{N}$ è prodotto di numeri primi. Dimostriamo adesso l'unicità della decomposizione. Sia $a = \prod_{i=1}^{l} p_i^{\alpha_i}$, p_i primi, $0 < p_i < p_{i+1}$, $\alpha_i \in \mathbb{N} \setminus \{0\}$. Supponiamo che esista un'altra decomposizione di $a = \prod_{j=1}^{m} q_j^{\beta_j}$, q_j primi, $0 < q_j < q_{j+1}$, $\beta_j \in \mathbb{N} \setminus \{0\}$. Sia p_1 un primo che divide a. Allora $p_1 | \prod_{j=1}^{m} q_j^{\beta_j}$ e quindi p_1 divide almeno un q_j. Ma p_1 e q_j sono

entrambi primi. Si ha allora $p_1 = q_j$. Dividiamo adesso a per p_1 e ripetiamo la procedura ottenendo che $p_k = q_r$ per una certa coppia di primi (che può anche essere uguale alla precedente a causa della molteplicità). Il Teorema segue iterando l volte il ragionamento. □

Ma quanti sono i numeri primi? Cominciamo col fornire una tabella di numeri primi "piccoli".

Esempio 1.1. *Tavola dei primi fino a* 2000 *:*

```
2   3   5   7   11  13  17  19  23  29  31  37  41  43  47  53
59  61  67  71  73  79  83  89  97  101 103 107 109 113
127 131 137 139 149 151 157 163 167 173 179 181
191 193 197 199 211 223 227 229 233 239 241 251
257 263 269 271 277 281 283 293 307 311 313 317
331 337 347 349 353 359 367 373 379 383 389 397
401 409 419 421 431 433 439 443 449 457 461 463
467 479 487 491 499 503 509 521 523 541 547 557
563 569 571 577 587 593 599 601 607 613 617 619
631 641 643 647 653 659 661 673 677 683 691 701
709 719 727 733 739 743 751 757 761 769 773 787
797 809 811 821 823 827 829 839 853 857 859 863
877 881 883 887 907 911 919 929 937 941 947 953
967 971 977 983 991 997 1009 1013 1019 1021 1031
1033 1039 1049 1051 1061 1063 1069 1087 1091 1093
1097 1103 1109 1117 1123 1129 1151 1153 1163 1171
1181 1187 1193 1201 1213 1217 1223 1229 1231 1237
1249 1259 1277 1279 1283 1289 1291 1297 1301 1303
1307 1319 1321 1327 1361 1367 1373 1381 1399 1409
1423 1427 1429 1433 1439 1447 1451 1453 1459 1471
1481 1483 1487 1489 1493 1499 1511 1523 1531 1543
1549 1553 1559 1567 1571 1579 1583 1597 1601 1607
1609 1613 1619 1621 1627 1637 1657 1663 1667 1669
1693 1697 1699 1709 1721 1723 1733 1741 1747 1753
1759 1777 1783 1787 1789 1801 1811 1823 1831 1847
1861 1867 1871 1873 1877 1879 1889 1901 1907 1913
1931 1933 1949 1951 1973 1979 1987 1993 1997 1999
```

Osservazione 1.1. *Osserviamo che nella tabella precedente sono presenti* 303 *primi minori di* 2000 *e che la loro distribuzione è inizialmente fitta e poi diviene più rada. Inoltre sono presenti* coppie *di primi (cioè primi dispari distanti* 2*) e si notano* buchi *relativamente grandi tra primi consecutivi (la spaziatura media è circa* 7*, la*

differenza massima è 34*). In definitiva, la distribuzione sembra* casuale *o perlomeno non risultano regolarità evidenti.*

Diamo ora qualche risposta alla domanda precedente. Prima di ciò osserviamo che, a parte il Teorema di Euclide 1.2 e la sua dimostrazione, talmente belli che non riusciamo a resistere alla tentazione di includerli, vogliamo anche fare un discorso di "densità" dei primi nella successione dei numeri naturali. In altre parole, vi sono molte successioni interessanti di interi, come i quadrati perfetti, le potenze di 2, i numeri di Fibonacci, e molte altre, e tutte queste, per la loro stessa definizione, hanno infiniti termini, ma se andiamo a contare il numero dei loro termini nell'intervallo $[1,N]$, con N grande, troviamo risultati piuttosto diversi. Alla luce del Teorema di Euclide, ha senso porsi questa domanda anche per la successione dei numeri primi.

Teorema 1.2 (Euclide). *Esistono infiniti numeri primi.*

Come accennavamo precedentemente, la dimostrazione del Teorema di Euclide è considerata una delle "gemme" della matematica antica; per tale ragione la riportiamo in questa sede.

Supponiamo dunque per assurdo che esista solamente un numero finito di primi e che questi siano esattamente $p_1 < p_2 < \cdots < p_k$. Allora il numero

$$N = p_1 p_2 \cdots p_k + 1$$

non può essere anch'esso primo (perché non è presente nell'elenco precedente); d'altronde, siccome N non è divisibile per alcun p_i, $i = 1,\dots,k$, deve anch'esso essere primo. Il che è chiaramente una contraddizione. Dunque l'insieme dei numeri primi non può essere finito e quindi il Teorema di Euclide è dimostrato.

Dopo aver dimostrato che esistono infiniti numeri primi è importante capire quale sia la loro densità negli interi. Il saper rispondere a questo problema non solo è significativo da un punto di vista teorico, ma ha anche riflessi applicativi poiché alcuni tra i più usati metodi crittografici moderni si basano sulla "facilità" di costruire numeri primi e sulla "difficoltà" di determinare la fattorizzazione di interi. Se i primi fossero "pochi" sarebbero difficili da costruire e di conseguenza il determinare la fattorizzazione di un intero dato risulterebbe facile.

Il primo passo nel capire quale fosse la densità dei primi fu fatto alla fine del XVIII secolo da Gauss il quale congetturò che il numero dei primi fino ad N, N molto "grande", fosse

$$\pi(N) \sim \frac{N}{\log N} \quad \text{per} \quad N \to +\infty,$$

dove

$$\pi(N) = |\{p \le N,\ p \text{ primo}\}|$$

e la notazione $F(N) \sim G(N)$ indica che il rapporto $F(N)/G(N)$ ha limite 1 quando N tende a $+\infty$. Il fatto sorprendente è che Gauss elaborò la propria congettura basandosi solamente sulle (scarne) informazioni fornite dalle tavole dei numeri primi disponibili all'epoca.

La congettura di Gauss fu dimostrata indipendentemente nel 1896 da Hadamard e de la Vallée Poussin (e dopo allora assunse il nome di Teorema dei Numeri Primi) usando l'idea fondamentale introdotta da Riemann nel 1858: studiare la distribuzione dei primi mediante l'analisi complessa. Definita la funzione

$$\mathrm{li}(N) = \int_2^N \frac{dt}{\log t},$$

l'enunciato preciso di questo teorema è il seguente

Teorema 1.3 (Teorema dei Numeri Primi).

$$\pi(N) \sim \mathrm{li}(N) \quad per \quad N \to +\infty.$$

È stato dimostrato che $\mathrm{li}(N)$ è una approssimazione migliore a $\pi(N)$ di quanto sia $N/\log N$, ossia che, per N sufficientemente grande, si ha

$$|\pi(N) - \mathrm{li}(N)| \leq \left|\pi(N) - \frac{N}{\log N}\right|.$$

Osserviamo inoltre che il primo termine dello sviluppo asintotico di $\mathrm{li}(N)$ è dato da $N/\log N$, come si può vedere integrando per parti, e che quindi si può effettivamente pensare all'uso di $\mathrm{li}(N)$ come di un enunciato più preciso del TNP.

Non ci soffermiamo sulle idee di Riemann e sul ruolo fondamentale che la sua funzione ζ ha avuto nello sviluppo della Teoria dei Numeri perché questo discorso ci porterebbe troppo lontano. Notiamo solamente che, da un punto di vista euristico, il Teorema dei Numeri Primi 1.3 consente di aspettarci che, per N abbastanza grande, esista un primo ogni $\log N$ interi circa, ossia che esistano "tanti" numeri primi.

Facciamo anche notare che la densità dei primi fino ad N è nettamente maggiore rispetto a quelle delle successioni menzionate nella prima parte di questo paragrafo; infatti nell'intervallo $[1,N]$ ci sono approssimativamente $\sqrt{N}$ quadrati perfetti, circa $\log_2 N$ potenze di 2 ed approssimativamente $\log_\Phi N$ numeri di Fibonacci, dove $\Phi = \frac{1+\sqrt{5}}{2}$ è il numero aureo.

1.2 Come generare numeri primi (il Crivello di Eratostene)

In questo paragrafo parliamo del problema di determinare tutti i numeri primi nell'intervallo $[1,N]$ dove N è un intero "grande". Un metodo possibile è quello di

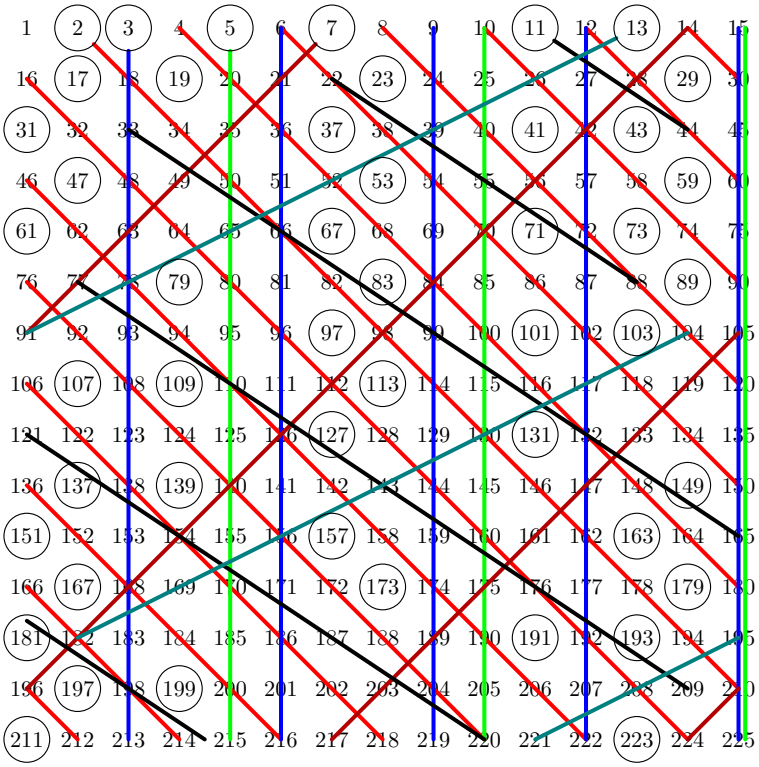

Figura 1.1: Il crivello di Eratostene. Il meccanismo di funzionamento è tutto sommato piuttosto semplice: saltato il numero 1, che non è primo, si prende il primo numero che soddisfa la Definizione 1.1, e cioè 2, lo si evidenzia mettendolo dentro un cerchio e se ne eliminano tutti i multipli fino al limite desiderato N, in questo caso 225. Si cerca quindi l'intero successivo non ancora cancellato, che è 3, lo si evidenzia e si eliminano tutti i suoi multipli. Si ripete la stessa procedura per tutti i numeri primi fino a $N^{1/2}$: ciò che resta sono il numero 1 e tutti i numeri primi nell'intervallo $[2,N]$. Le linee "cancellano" i multipli dei numeri primi 2, 3, 5, 7, 11 e 13.

scorrere la lista di tutti questi interi, e verificarne singolarmente l'eventuale primalità: Eratostene scoprí che questa procedura è estremamente dispersiva, nel senso che gran parte del calcolo è inutilmente ripetuta più volte, e quindi suggerí una strategia alternativa che risulta di gran lunga più efficiente.

La procedura di Eratostene è nota con il nome di "crivello" (che vuol dire "setaccio"): si passano i numeri interi positivi attraverso un opportuno setaccio, e quelli che restano sono solo i numeri primi. La Figura 1.1 illustra il Crivello di Eratostene applicato all'intervallo $[1,225]$. È relativamente semplice spiegare il procedimento in forma algoritmica: supponiamo di voler eseguire il crivello sugli interi nell'intervallo $[1,N]$, dove N è un parametro a nostra scelta.

(1) Si scrivono tutti gli interi da 1 ad N.

(2) Si parte da $p = 2$ (il più piccolo numero primo).

(3) Si cancellano tutti i multipli di p partendo da p^2 fino ad N.

(4) Si cerca il più piccolo intero $q > p$ non ancora cancellato. Se $q^2 > N$ la procedura termina.

(5) Si pone $p = q$ e si torna al passo (3).

Si noti che l'operazione di cancellazione di cui al punto (3) può essere effettuata in modo estremamente efficiente (ed è esattamente qui l'essenza dell'algoritmo) partendo da p^2 ed aggiungendo sempre p all'ultimo valore trovato, fino a superare N. Infatti i multipli di p fra $2p$ e $p^2 - p$, estremi inclusi, sono stati cancellati tutti nei passi precedenti, poiché hanno almeno un fattore primo $< p$. Si veda anche la Figura 1.2.

A questo punto il nostro obiettivo è dimostrare che i numeri "sopravvissuti" a questa operazione sono il numero 1 e tutti i numeri primi fino ad N. Il numero 1, evidentemente, non è stato mai cancellato (il più piccolo numero cancellato è 4, alla prima iterazione). I numeri primi nell'intervallo $[1,N^{1/2}]$ non sono stati cancellati, perché il più piccolo multiplo del primo p che è effettivamente cancellato è $2p$, e i numeri primi nell'intervallo $[N^{1/2},N]$ non sono stati cancellati perché non hanno divisori fra i numeri con i quali abbiamo effettuato le cancellazioni.

Infine, tutti i numeri composti nell'intervallo $[1,N]$ sono stati cancellati: infatti, sia $n \leq N$ un numero composto. Dunque esistono interi a, b con $1 < a \leq b < n$ e tali che $n = ab$. Se a e b fossero entrambi $> N^{1/2}$, il loro prodotto n dovrebbe essere a sua volta $> N$. Dunque n è divisibile per almeno un numero primo $\leq N^{1/2}$ (ogni fattore primo di a va bene), e di conseguenza è stato eliminato.

Invitiamo i Lettori a convincersi della correttezza dell'algoritmo eseguendo materialmente queste operazioni sui numeri da 1 a 100: conviene prima disporre i

numeri in uno schema quadrato come nella Figura 1.1. Si noterà come i multipli di 2, 3, 5, 7 compaiono lungo opportuni segmenti.

Questo è lo schema di base: naturalmente, quando si esegue questa procedura al computer, non è necessario "scrivere" esplicitamente gli interi fra 1 ed N, e sono anche possibili alcuni miglioramenti che ora andiamo a discutere. Questo schema può essere scritto in "pseudo-codice" (cioè in una forma intermedia fra il linguaggio naturale ed un linguaggio di programmazione vero e proprio) e proponiamo la nostra versione nella Figura 1.2. Le istruzioni nelle righe 6–9 realizzano la fase di "cancellazione" mentre le righe 10–12 eseguono la ricerca del primo intero non cancellato successivo a p.

Se intendiamo scrivere una procedura per computer che esegua il Crivello di Eratostene, dobbiamo cominciare con l'assegnare un certo spazio di memoria per rappresentare i numeri dell'intervallo $[1,N]$: normalmente questo viene realizzato creando un `array` (cioè una matrice ad una dimensione, altrimenti detto vettore) con N posizioni. Dato che quello che ci interessa è sapere se il numero intero corrispondente alla k-esima posizione di questo array è primo oppure no, questo array conterrà inizialmente il valore `vero` in tutte le sue N posizioni (si veda la riga 2), e l'operazione di *cancellazione* descritta nel passo 3 corrisponderà a trasformare questo valore in `falso` (riga 7). In linguaggio più tecnico, abbiamo un array di variabili *logiche* (dette anche `booleani`), e possiamo pensare che il valore `vero` corrisponda, alla fine del calcolo, ai numeri primi ed al numero 1.

La prima cosa che si nota è che c'è un certo "spreco": infatti, tutte le posizioni dell'array corrispondenti ad interi pari ≥ 4 contengono numeri che certamente non sono primi, e quindi stiamo sprecando metà circa dell'array per contenere informazioni inutili, e per sovrammercato stiamo anche sprecando del tempo per ottenere queste stesse informazioni!

Dunque, cerchiamo un modo per sfruttare questo fatto, ed utilizzare la metà circa delle posizioni necessarie seguendo alla lettera la "ricetta" data qui sopra. Tutto si basa sul Lemma seguente, che è la generalizzazione dell'osservazione, in sé piuttosto banale, che tutti i numeri primi, a parte il numero 2, sono dispari.

Lemma 1.1. *Sia $M \geq 1$ un numero intero. Se p è un numero primo, allora $p \mid M$ oppure $(p,M) = 1$.*

La prima modifica che possiamo apportare allo schema dato sopra è questa: se abbiamo N posizioni dell'array, invece di far corrispondere la k-esima posizione al numero k (dove k assume i valori 1, 2, 3, ..., N), la facciamo corrispondere al numero $2k-1$. Questo significa che utilizzando N posizioni, siamo in grado di eseguire il Crivello sugli interi da 1 a $2N-1$, oppure, che possiamo eseguire il crivello sugli interi da 1 ad N utilizzando circa $N/2$ posizioni. In entrambi i casi, abbiamo un risparmio del 50% circa.

IL CRIVELLO DI ERATOSTENE, VERSIONE DI BASE

```
 1 function crivello(N)
 2 boolean c[N] = {vero * N}
 3 int p ← 2
 4 while (p^2 ≤ N)
 5       int n ← p^2
 6       while (n ≤ N)
 7             c[n] ← falso
 8             n ← p + n
 9       endwhile
10       repeat
11             p ← p + 1
12       until (c[p] = vero)
13 endwhile
```

Figura 1.2: Lo pseudo-codice per il Crivello di Eratostene.

Dobbiamo subito notare che il passo 2 deve essere sostituito da "Si parte da $p = 3$" (in un certo senso, il passo con $p = 2$ è implicito nella nostra costruzione), che l'operazione di cancellazione dei multipli di p deve tener conto del fatto che p^2 (il primo intero da cancellare) corrisponde alla posizione $\frac{1}{2}(p^2+1)$, e che non vi sono multipli pari da cancellare, ma solo quelli dispari. Questo comporta una certa complicazione nei dettagli della realizzazione pratica, che qui omettiamo.

Il discorso appena fatto si riferisce al caso $M = 2$ del Lemma 1.1, ma vi sono valori più grandi di M per cui il rapporto tra risparmio e complicazione nei dettagli rimane vantaggioso. Ne descriviamo uno particolarmente interessante.

Se scegliamo $M = 30$ nel Lemma 1.1, tutti i numeri primi, a parte 2, 3, 5, non hanno fattori comuni con 30, e quindi giacciono in una delle progressioni aritmetiche $1+30k$, $7+30k$, $11+30k$, $13+30k$, $17+30k$, $19+30k$, $23+30k$, $29+30k$. La cosa interessante dal punto di vista "informatico" è che il numero di queste progressioni è 8, e quindi possiamo pensare di associare ogni intervallo di 30 interi consecutivi ad un byte, (diciamo che associamo l'intervallo di estremi $30k$ e $30(k+1)$ al k-esimo byte per $k \in \mathbb{N}$), e associamo il j-esimo bit del byte in questione all'intero $a_j + 30k$, dove gli a_j, in ordine, valgono 1, 7, 11, 13, 17, 19, 23, 29, quando j va da 0 a 7. In questo modo non c'è "spreco" e si usano i singoli bit, con la convenzione che `vero` corrisponde al valore 1 e `falso` al valore 0.

Ovviamente il numero 30 non è speciale, ed è possibile usare, per esempio, $N = 210$, nel quale caso vi sono 48 progressioni da esaminare e la faccenda diventa più complicata.

Come osservazione finale, notiamo che sostanzialmente la stessa procedura funziona se si vogliono determinare i numeri primi nell'intervallo $[M, M+N]$, dove M è un intero grande. C'è una certa complicazione nel dettaglio, dovuta al fatto che il primo intero da eliminare non è necessariamente p^2 come nel nostro schema, ma l'idea di base rimane la stessa.

Questa osservazione ha rilevanza pratica: se si vuole determinare un numero primo "grande" p, diciamo dell'ordine di grandezza di M, si sceglie N dell'ordine di grandezza di $2\log M$ (in modo che vi sia una ragionevole speranza che l'intervallo $[M, M+N]$ contenga almeno un numero primo), e si opera un crivello con tutti i numeri primi relativamente piccoli. Evidentemente, questo non garantisce che i numeri sopravvissuti al crivello siano effettivamente primi, ma in questo modo si eliminano dall'intervallo in questione, in modo molto efficiente, tutti i numeri che non hanno alcuna speranza di essere primi. A questo punto, si sottopongono i numeri rimasti, che sono relativamente pochi, a dei test di primalità (si vedano, per esempio, il libro [6] oppure [7] o gli articoli divulgativi [9] e [10]), che sono più onerosi dal punto di vista computazionale.

In definitiva, il vantaggio risiede nel fatto che la stragrande maggioranza degli interi vengono eliminati con un basso "costo unitario" e i pochi interi residui possono essere attaccati singolarmente, ad un costo individuale maggiore.

Osservazione 1.2. *Al 07 gennaio 2016 il più grande primo conosciuto è un numero è un numero di Mersenne dato da*

$$2^{74\,207\,281} - 1$$

che ha 22 338 618 *cifre decimali. Si veda* `http://www.mersenne.org/`.

Nota Bene 1.1. *Abbiamo deciso di non sviluppare alcun metodo di primalità e di presentare solamente il Crivello di Eratostene perché anche il più semplice test di primalità oggi noto richiede un'analisi accurata di quanti sono i numeri composti che verificano il Piccolo Teorema di Fermat 1.7 (tali composti, visto che si comportano come se fossero primi, vengono detti* pseudoprimi*). In metodi un poco più sofisticati è necessario conoscere quante e quali sono le radici quadrate di un dato $a \in \mathbb{Z}$ modulo n, dove n è il numero di cui si vuole verificare la primalità. Una presentazione di questi metodi si può, ad esempio, trovare in [6] o in [7]. Il Crivello di Eratostene, sebbene risponda da una domanda diversa (la costruzione di una "tabella" di primi), ha il pregio di essere totalmente elementare e di avere una buona complessità "media"; si veda l'Incontro 3.1.*

1.3 Il Massimo Comun Divisore e l'Algoritmo Euclideo

Passiamo adesso a presentare un algoritmo ben conosciuto che serve a determinare in modo "veloce" il massimo comun divisore di due interi (nel seguito denotato da (a,b)).

Algoritmo 1 (Euclideo). *Dati $a,b \in \mathbb{N}$, $a > b$, vogliamo calcolare (a,b).*

(1) dividiamo a per b: ottengo $a = bq_1 + r_1$; $0 \leq r_1 < b$;

(2) dividiamo b per r_1; ottengo $b = r_1q_2 + r_2$; $0 \leq r_2 < r_1$;

(3) dividiamo r_1 per r_2; ottengo $r_1 = r_2q_3 + r_3$; $0 \leq r_3 < r_2$;

(4) e così via ottenendo all'ultimo passo: $r_{k-2} = r_{k-1}q_k$; ossia $r_k = 0$.

(5) Allora

$$r_{k-1} = (a,b).$$

L'algoritmo termina quando si ottiene un resto che divide il resto precedente perché gli r_i sono una successione strettamente decrescente di interi positivi o nulli ed una tale successione può avere solo finiti elementi.

Il fatto che $r_{k-1} = (a,b)$ dipende da:

(1) ogni divisore comune di a,b divide anche ogni r_i, $i = 1,\ldots,k-1$;

(2) tutti gli interi $a,b,r_1,\ldots,r_{k-2}$ sono multipli di r_{k-1}.

Allora r_{k-1} è il massimo comun divisore perché è l'unico numero che divide sia a che b e che è divisibile per ogni divisore comune di a e b.

Esempio 1.2. *Calcoliamo $(1547,560)$ utilizzando l'Algoritmo Euclideo:*

$$\begin{array}{rcrcr} 1547 & = & 2\cdot 560 & + & 427 \\ 560 & = & 1\cdot 427 & + & 133 \\ 427 & = & 3\cdot 133 & + & 28 \\ 133 & = & 4\cdot 28 & + & 21 \\ 28 & = & 1\cdot 21 & + & 7 \\ 21 & = & 3\cdot 7 & + & 0. \end{array}$$

e quindi $(1547,560) = 7$.

Abbiamo detto precedentemente che l'Algoritmo Euclideo è "veloce". Ma come si misura la velocità di un algoritmo? Il parametro che si sceglie di utilizzare è quello del *numero di operazioni elementari* compiute. La valutazione di tale quantità per un certo algoritmo viene detta *complessità computazionale* dell'algoritmo stesso. Poiché il nostro termine di paragone è l'utilizzo sui computer indicheremo come operazione elementare *una operazione fatta su una cifra binaria* intendendo in tal modo l'operazione che, dati tre bit, ne fornisce la somma (bisogna considerare la presenza del bit di riporto).

In tal modo è immediato osservare che, se si sommano due interi n, m, con $m < n$, allora si eseguiranno al più $c \log n$ operazioni elementari ($c > 0$ è una costante opportuna e $\log n$ indica essenzialmente il numero di cifre di n). In tal caso si dice che

l'operazione di somma ha complessità computazionale $O(\log n)$.

La notazione O-grande è molto comoda quando si devono dare stime dall'alto (maggiorazioni) di una certa funzione: avremo che

$$f(n) = O(g(n)) \quad \text{se e solo se esiste una costante } c > 0 \text{ tale che } |f(n)| \le c|g(n)|$$

per tutti gli n sufficientemente grandi.

Torniamo all'Algoritmo Euclideo. Ricordiamo che abbiamo a, b naturali con $a > b$ e che vogliamo calcolare (a, b). Per stimarne la complessità computazionale ci serve la

Proposizione 1.1. *I resti dell'Algoritmo Euclideo verificano* $r_{j+2} < \frac{1}{2} r_j$.

Dim. Supponiamo che valga $r_{j+1} \le \frac{1}{2} r_j$. Allora abbiamo $r_{j+2} < r_{j+1} \le \frac{1}{2} r_j$. Supponiamo adesso $r_{j+1} > \frac{1}{2} r_j$. Dal passo successivo dell'Algoritmo Euclideo otteniamo $r_j = 1 \cdot r_{j+1} + r_{j+2}$ e quindi $r_{j+2} = r_j - r_{j+1} < \frac{1}{2} r_j$. □

Allora ogni due passi dell'Algoritmo Euclideo i resti vengono dimezzati. Poiché il resto non può mai essere più piccolo di 1 e $a > b$, si possono al più fare

$$2 \cdot \lfloor \log_2 a \rfloor \quad \text{divisioni di numeri } \le a.$$

Ma, ricordando il metodo elementare con cui moltiplicazioni e divisioni vengono ricondotte all'operazione di somma e che tutti i numeri coinvolti sono $\le a$, abbiamo immediatamente che

ogni divisione o moltiplicazione ha complessità $O\big(\log^2 a\big)$.

Da ciò segue che *la complessità computazionale dell'Algoritmo Euclideo è* minore di $O\big(\log^3 a\big)$ operazioni elementari.

Osservazione 1.3. *In realtà, effettuando una stima più accurata, si può dimostrare che la complessità dell'Algoritmo Euclideo è* $O\big(\log^2 a\big)$ *operazioni elementari.*

Come ultimo argomento di questo paragrafo proviamo la seguente utile relazione tra due interi ed il loro massimo comun divisore.

Proposizione 1.2 (Formula di Bézout). *Sia $d = (a,b)$, $a > b$. Allora esistono*

$$u, v \in \mathbb{Z} \text{ tali che } d = ua + vb$$

e tali u, v possono essere determinati con complessità $O(\log^3 a)$.

Dim. Seguendo a ritroso le uguaglianze dell'Algoritmo Euclideo e scrivendo ogni volta d in termini dei resti precedenti, otteniamo la tesi. Ad ogni passo del procedimento si eseguono un'addizione ed una moltiplicazione; quindi la complessità di ogni passo è dominata dalla complessità della moltiplicazione. □

Esempio 1.3. *Verifichiamo la Formula di Bézout con i dati dell'esempio precedente:*

$$\begin{array}{rcll} 7 & = & 28 - 1\cdot 21 & = & 28 - 1\cdot(133 - 4\cdot 28) \\ & = & 5\cdot 28 - 1\cdot 133 & = & 5(427 - 3\cdot 133) - 1\cdot 133 \\ & = & 5\cdot 427 - 16\cdot 133 & = & 5\cdot 427 - 16(560 - 1\cdot 427) \\ & = & 21\cdot 427 - 16\cdot 560 & = & 21\cdot(1547 - 2\cdot 560) - 16\cdot 560 \\ & = & 21\cdot 1547 - 58\cdot 560 & & \end{array}$$

da cui segue $u = 21$ e $v = -58$.

Nota Bene 1.2. *Una versione dell'Algoritmo Euclideo con Formula di Bézout in pseudocodice ed uno script PARI/Gp sono presentati nell'Incontro 3.2.*

1.4 Teoria delle Congruenze

Nel seguito utilizzeremo spesso la seguente:

Definizione 1.2. *Dati tre interi a, b, n diremo che a è* congruente a b modulo n *(e scriveremo $a \equiv b \bmod n$) se e solo se n divide $a - b$ (e scriveremo $n|(a-b)$).*

Si noti che, siccome il resto r della divisione di a per n è tale che $0 \leq r \leq n-1$, allora ogni intero a è congruente modulo n ad uno degli interi

$$0, 1, 2, \ldots, n-1.$$

Inoltre è immediato osservare che due interi sono congruenti modulo n se e solo se i loro resti per la divisione per n sono uguali. Quindi due interi congruenti modulo n vengono "mappati" mediante divisione per n nello stesso intero $r \in \{0, 1, 2, \ldots, n-1\}$.

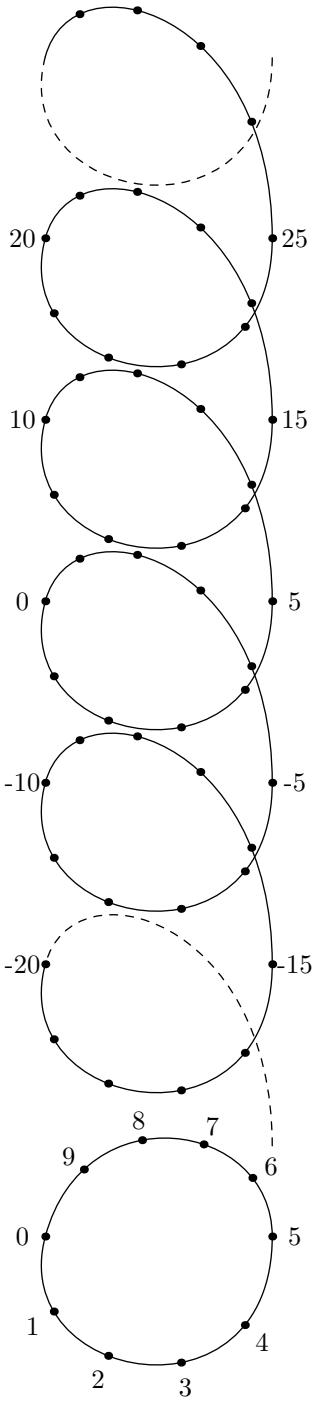

Figura 1.3: Si noti come sia possibile "pensare" a $\mathbb{Z}$ che si "proietta" su $\mathbb{Z}_{10}$. Essenzialmente $\mathbb{Z}_{10}$ è dato dai punti segnati sulla circonferenza di base e ogni altra successiva o precedente decina di elementi di $\mathbb{Z}$ si può pensare come se appartenesse ad un'elica "costruita" sulla circonferenza di partenza.

Tale insieme è anche detto l'insieme delle *classi residuali modulo n* e viene denotato $\mathbb{Z}_n$ (anello degli interi modulo n). In un certo senso $\mathbb{Z}$ "collassa" su $\mathbb{Z}_n$; si veda ad esempio la Figura 1.3.

Esaminiamo adesso le operazioni in $\mathbb{Z}_n$.

Proposizione 1.3. *Siano $a \equiv b \bmod n$ e $c \equiv d \bmod n$. Allora $a+c \equiv b+d \bmod n$, $a-c \equiv b-d \bmod n$, $ac \equiv bd \bmod n$.*

Dim. Dopo aver osservato che $a \equiv b \bmod n$ equivale a dire che esiste un intero k opportuno per cui $a = b+nk$, la dimostrazione è immediata. □

Purtroppo per la divisione la questione è più problematica.

Proposizione 1.4. *Sia $ac \equiv bc \bmod n$. Allora*

$$a \equiv b \bmod \frac{n}{(c,n)}.$$

Dim. Sia $d = (c,n)$. Allora, poiché $n|(a-b)c$, si ha $\frac{n}{d}|(a-b)\frac{c}{d}$. Ma $(\frac{c}{d}, \frac{n}{d}) = 1$ e quindi $\frac{n}{d}|(a-b)$. □

Quindi entrambi i lati di una congruenza possono essere divisi per un intero c a patto che contemporaneamente il modulo della congruenza stessa venga diviso per (c,n).

Le regole dell'aritmetica modulo n sono molto comode perché consentono di lavorare con numeri "piccoli".

Esempio 1.4. *(1) Calcoliamo $3^{100} \bmod 101$. Per l'elevazione a potenza utilizziamo la tecnica dei Quadrati Ripetuti (un algoritmo che realizza i Quadrati Ripetuti si trova nell'Appendice 3.6 degli Incontri):*

$$\begin{array}{llll}
3^5 & = 243 & \equiv 41 \bmod 101 & \\
3^{10} & = (3^5)^2 & \equiv (41)^2 & = 1681 \quad \equiv 65 \bmod 101 \\
3^{20} & = (3^{10})^2 & \equiv (65)^2 & = 4225 \quad \equiv -17 \bmod 101 \\
3^{40} & = (3^{20})^2 & \equiv (-17)^2 & = 289 \quad \equiv -14 \bmod 101 \\
3^{50} & = 3^{40}3^{10} & \equiv (-14)\cdot 65 & = -910 \quad \equiv -1 \bmod 101 \\
3^{100} & = (3^{50})^2 & \equiv (-1)^2 & = 1 \quad \equiv 1 \bmod 101.
\end{array}$$

Come vedremo in seguito tale risultato è in realtà una conseguenza del Piccolo Teorema di Fermat 1.7 nella sua forma forte.

(2) Calcoliamo $4^n \bmod 12$ per ogni $n \in \mathbb{N}$, $n \geq 1$. Abbiamo

$$\begin{array}{llll}
4^2 & = 16 & \equiv 4 \bmod 12 & \\
4^3 & = 4(4)^2 & \equiv 16 & \equiv 4 \bmod 12
\end{array}$$

e così via ottenendo infine

$$4^n \equiv 4 \bmod 12$$

per ogni $n \in \mathbb{N}$, $n \geq 1$.

Osserviamo che in realtà ciò segue nuovamente dal Piccolo Teorema di Fermat 1.7 integrato da un piccolo ragionamento sulla divisibilità. Infatti esso implica che $4^n \equiv 4 \bmod 3$. *Inoltre è ovvio che* $4^n \equiv 0 \equiv 4 \bmod 4$. *Quindi* $3|(4^n-4)$ *e* $4|(4^n-4)$. *Siccome* $(3,4)=1$, *si ha che* $12|(4^n-4)$ *cioè* $4^n \equiv 4 \bmod 12$. *Analoghi ragionamenti sulla divisibilità si useranno nella dimostrazione del Teorema Cinese del Resto 1.5.*

Applichiamo adesso l'Algoritmo Euclideo al calcolo degli elementi invertibili di $\mathbb{Z}_n$.

Proposizione 1.5. *Gli elementi* a *di* $\mathbb{Z}_n$ *che hanno inverso moltiplicativo sono tutti e soli quelli per cui* $(a,n)=1$. *L'inverso* b *di* a *in* $\mathbb{Z}_n$ *si può calcolare con complessità* $O\big(\log^3 n\big)$.

Dim. Sia $d=(a,n)$. Se $d>1$ non può esistere un b tale che $ab \equiv 1 \bmod n$ perché si avrebbe $d|(ab-1)$ e quindi, siccome $d|a$, si avrebbe $d|1$ che è contraddittorio con l'assunzione $d>1$. Allora $d=1$. Supponiamo, senza ledere la generalità, che $a<n$. Utilizzando l'Algoritmo Euclideo, sappiamo che esistono $u,v \in \mathbb{Z}$ tali che $ua+vn=1$ e che u,v sono calcolabili con complessità $O\big(\log^3 n\big)$. L'inverso di a è allora $b=u$. Infatti si ottiene che $n|(1-ua)$ cioè $ua \equiv 1 \bmod n$. □

Diamo ora la seguente

Definizione 1.3. *L'insieme degli elementi di* $\mathbb{Z}_n$ *che possiedono inverso moltiplicativo viene denotato* $\mathbb{Z}_n^*$. *Quindi*

$$\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n \ : \ (a,n)=1\}.$$

1.4.1 Congruenze lineari

Siano a,b due note classi residuali modulo n e x una classe residuale incognita tale che

$$ax \equiv b \bmod n. \tag{1.1}$$

Tale congruenza equivale alla equazione diofantea $ax=b+ny$ che è risolubile (negli interi) se e solo se $(a,n)|b$.

Inoltre, se $(a,n)=d$, la (1.1) equivale alla $\frac{a}{d}x \equiv \frac{b}{d} \bmod \frac{n}{d}$ in cui $(\frac{a}{d},\frac{n}{d})=1$. Quindi il problema si riduce a studiare $a'x \equiv b' \bmod n'$ con $(a',n')=1$. Per tale caso abbiamo il seguente

Teorema 1.4. *La congruenza lineare* $ax \equiv b \bmod n$ *con* $(a,n) = 1$ *ammette un'unica soluzione data da* $x \equiv a^{-1}b \bmod n$.

Dim. Sia $x \in \mathbb{Z}_n$. Allora tutti i valori di $ax \bmod n$ sono distinti (perché $ax_1 \equiv ax_2 \bmod n$ implica $x_1 \equiv x_2 \bmod n$ essendo $(a,n) = 1$). Poiché esistono esattamente n classi residuali, gli n differenti valori di $ax \bmod n$ devono appartenere ad n classi residuali distinte. Allora per esattamente un valore di $x \bmod n$ esiste uno specifico valore b per cui la congruenza $ax \equiv b \bmod n$ è verificata. Tale valore è $a^{-1}b$ perché $(a,n) = 1$ implica che a ammette inverso moltiplicativo. □

Analogamente a quanto avviene per i sistemi di equazioni, è possibile studiare le soluzioni di sistemi di congruenze lineari. Si consideri il sistema

$$\begin{cases} a_1 x \equiv b_1 \bmod n_1 \\ \dots \\ a_k x \equiv b_k \bmod n_k \end{cases} \qquad \text{con } (a_i, n_i) = 1,\ i = 1, \dots, k \tag{1.2}$$

in cui abbiamo assunto che ogni congruenza lineare sia risolubile. Abbiamo il

Teorema 1.5 (Cinese del Resto). *Nell'ipotesi che* $(n_i, n_j) = 1$, $i \neq j$, *il sistema* (1.2) *ammette un'unica soluzione* $x \bmod \prod_{i=1}^{k} n_i$.

Dim. Supponiamo $k = 2$. Dalla prima congruenza abbiamo che esiste $t \in \mathbb{Z}$ tale che $a_1 x = b_1 + n_1 t$. Moltiplicando quest'ultima per n_2 otteniamo $n_2 a_1 x = n_2 b_1 + n_2 n_1 t$. Ragionando analogamente, dalla seconda congruenza segue che esiste $u \in \mathbb{Z}$ tale che $n_1 a_2 x = n_1 b_2 + n_1 n_2 u$. Sottraendo le due equazioni ottenute si ha $(n_2 a_1 - n_1 a_2)x = n_2 b_1 - n_1 b_2 + n_2 n_1 (t - u)$, cioè

$$(n_2 a_1 - n_1 a_2)x \equiv n_2 b_1 - n_1 b_2 \bmod n_1 n_2.$$

Si noti che questa congruenza è risolubile perché, dato che $(n_2 a_1 - n_1 a_2, n_1) = (n_2 a_1, n_1) = 1$ e $(n_2 a_1 - n_1 a_2, n_2) = (n_1 a_2, n_2) = 1$, si ha $(n_2 a_1 - n_1 a_2, n_1 n_2) = 1$ grazie all'ipotesi $(n_1, n_2) = 1$. Infine, la soluzione è unica modulo $n_1 n_2$ per il Teorema 1.4. Il caso generale $k \geq 3$ si risolve raccogliendo le congruenze a coppie e riconducendosi al risultato $k = 2$. □

Osservazione 1.4. *Quella precedente è una delle tante possibili trattazioni del Teorema 1.5. Un'altra possibilità è, ad esempio, quella di dimostrare che, se* $(m,n) = 1$, *tra* $\mathbb{Z}^*_{mn}$ *e* $\mathbb{Z}^*_m \times \mathbb{Z}^*_n$ *esiste una biiezione data da* $a + mn\mathbb{Z} \mapsto (a + m\mathbb{Z}, a + n\mathbb{Z})$.

1.5 Piccolo Teorema di Fermat

Della Proposizione 1.5 sappiamo che gli elementi invertibili di $\mathbb{Z}_n$ sono tutti e soli gli elementi coprimi con n stesso. Chiaramente se $n = p$ è un numero primo, allora

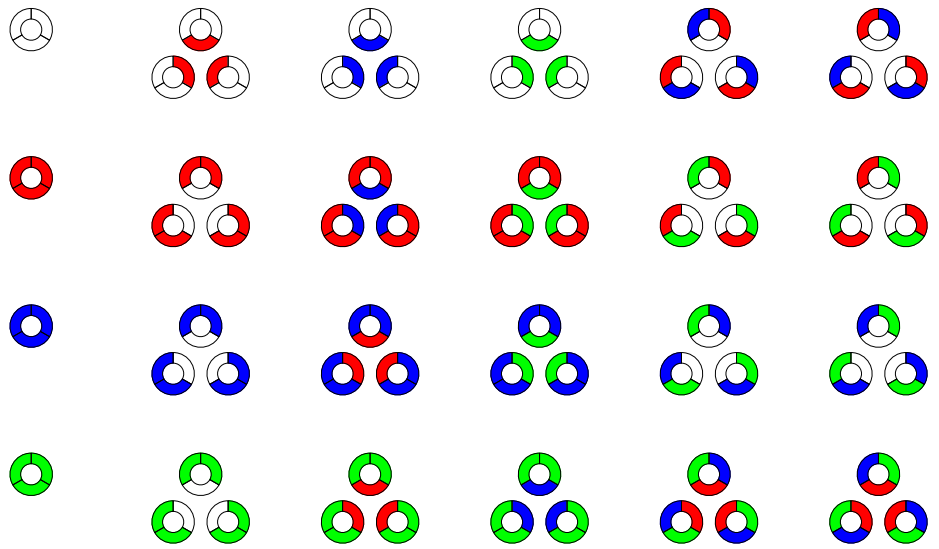

Figura 1.4: Dimostrazione del Piccolo Teorema di Fermat: le 64 collane con 3 perline di 4 colori; quelle policrome sono raggruppate in classi di collane equivalenti per rotazione.

ogni elemento di $\mathbb{Z}_p$ diverso da zero ha inverso moltiplicativo. Abbiamo cioè il seguente

Teorema 1.6. *Sia p primo. Allora $\mathbb{Z}_p$ è un campo (ossia $\mathbb{Z}_p^* = \mathbb{Z}_p \setminus \{0\}$).*

Tornando all'esame delle soluzioni delle congruenze lineari, osserviamo che, nel caso in cui il modulo della congruenza sia un numero primo p, il problema della cancellazione in $ac \equiv bc \bmod p$ diviene particolarmente interessante. Infatti possiamo avere solo due casi: $(c,p) = p$ oppure $(c,p) = 1$. Se $(c,p) = p$ il caso è banale perché si ottiene $a \equiv b \bmod 1$ che è una congruenza verificata da ogni $a, b \in \mathbb{Z}$. Se $(c,p) = 1$ possiamo ridurre entrambi i lati *senza* toccare il modulo cioè otteniamo che la congruenza iniziale equivale a $a \equiv b \bmod p$.

Esercizio 1.1. *Si studi la struttura moltiplicativa del sottoinsieme di $\mathbb{Z}_5$ formato da tutti i suoi elementi invertibili.*

Presentiamo adesso un teorema che risulterà di importanza fondamentale nel seguito.

Teorema 1.7 (Piccolo Teorema di Fermat). *Sia a un intero qualsiasi, e p un numero primo. Allora $a^p \equiv a \bmod p$.*

Dim. Consideriamo tutte le *collane* con p perline, ciascuna delle quali può avere uno qualsiasi fra a colori diversi. In particolare riterremo *ordinate* tali collane; os-

sia due collane con p perline sono diverse se differiscono o per il colore delle perline usate o per l'ordinamento delle perline stesse. Con $p = 3$, ad esempio, la collana (rosso, nero, bianco) è diversa da quella (bianco, nero, rosso). Vi sono evidentemente a^p collane possibili, a delle quali sono monocromatiche. Suddividiamo le rimanenti $a^p - a$ collane (policrome, cioè con perline di almeno due colori diversi) in classi di equivalenza, come segue: due collane sono equivalenti se una si ottiene dall'altra mediante un'opportuna rotazione del piano. Evidentemente ogni classe non può contenere più di p collane fra loro equivalenti, ma, poiché p è primo, per il Lemma 1.2 ogni classe ne deve contenere esattamente p. Dunque, p divide $a^p - a$. □

La Figura 1.4 illustra la dimostrazione nel caso $p = 3$, $a = 4$.

Lemma 1.2. *Se p è un numero primo, nessuna collana policroma con p perline è uguale ad una propria rotazione non banale.*

Dim. Supponiamo che la collana data sia uguale alla propria rotazione in senso orario di r perline, con $r > 1$ (altrimenti la collana sarebbe monocromatica) ed $r < p$ (altrimenti la rotazione sarebbe banale). Fissiamo r perline consecutive. Le r perline successive a quelle fissate devono formare un blocco uguale alle r precedenti (dato che dopo la rotazione vanno a coincidere). In altre parole, la collana è formata da due blocchi identici consecutivi di r perline ciascuno, seguite da $p - 2r$ perline. Possiamo ripetere lo stesso identico ragionamento per le ulteriori r perline successive, che devono a loro volta formare un altro blocco uguale ai due precedenti, dando luogo a 3 blocchi identici consecutivi seguiti da $p - 3r$ perline. Iterando questo procedimento fino ad esaurire tutte le perline della collana senza tralasciarne nessuna, troviamo che questa è costituita da k blocchi di r perline ciascuno. In definitiva, $r \mid p$ e p non è un numero primo. □

È opportuno notare che, se p è primo, la rotazione di una collana produce una collana *diversa*, mentre questo non è vero se p non è primo, come si vede dal caso con $p = 20$ illustrato dalla Figura 1.5.

Per esempio, abbiamo $2^{91} \equiv 37 \bmod 91$, e quindi 91 non è un numero primo. Questa è solo una dimostrazione "indiretta" del fatto che 91 non è primo, ed infatti non ne vengono ricavati i fattori primi. La situazione è anche complicata dal fatto che la congruenza di Fermat dà solo una condizione *necessaria* per la primalità di un intero. In effetti, $3^{91} \equiv 3 \bmod 91$, nonostante il fatto che 91 non sia un numero primo.

Osservazione 1.5. *Questo è il punto di partenza di un metodo di pseudoprimalità. Ossia un algoritmo che, se risponde che n è composto lo fa correttamente, mentre anche se n ha passato la verifica non può concludere che n è primo (perché esistono numeri composti che verificano la proprietà in questione). Un'analisi dettagliata*

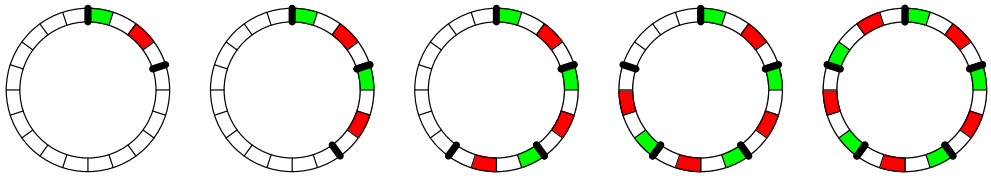

Figura 1.5: La dimostrazione del Lemma 1.2. La collana è invariante per una rotazione in senso orario di 4 perline, e quindi, scelte in modo arbitrario 4 perline consecutive, a sinistra, sappiamo che le 4 perline immediatamente adiacenti, sempre in verso orario, devono essere colorate allo stesso modo. Per lo stesso motivo, anche le successive 4 hanno la stessa colorazione, e cosí via: il numero totale di perline deve essere un multiplo di 4, e quindi non è primo.

di questi fatti ci porterebbe troppo lontano; chi vuole può leggere questi (ed altri) ragionamenti sul libro di Koblitz [5] oppure su [6] o [7].

Dal Teorema 1.6 sappiamo che gli elementi invertibili di $\mathbb{Z}_p$ sono tutti e soli i suoi elementi non nulli. Sia $a \not\equiv 0 \bmod p$. Moltiplicando allora ogni lato dell'equazione $a^p \equiv a \bmod p$ per a^{-1} otteniamo $a^{p-1} \equiv 1 \bmod p$ che viene anche detta *forma forte del Piccolo Teorema di Fermat*. Questa seconda forma (ed anche la prima, in realtà) può essere dimostrata in maniera più formale, come vediamo nel paragrafo seguente.

1.5.1 Dimostrazione alternativa del Piccolo Teorema di Fermat

Proviamo il fatto che se p primo, $a \not\equiv 0 \bmod p$, allora

$$a^{p-1} \equiv 1 \bmod p.$$

Dim. Notiamo che $\{0a, 1a, 2a, \dots, (p-1)a\}$ è un insieme di residui completo modulo p. Infatti se così non fosse due di essi distinti ia, ja starebbero nella stessa classe residuale modulo p cioè $ia \equiv ja \bmod p$. Ma allora $p|(i-j)a$ che implica $p|(i-j)$. Ciò non è possibile perché $i < p$ e $j < p$; quindi $i = j$. Possiamo quindi affermare che $0a, 1a, 2a, \dots, (p-1)a$ è solamente un riarrangiamento di $0, 1, 2, \dots, (p-1)$. Inoltre si ha certamente che $0a = 0$ per ogni $a \in \mathbb{Z}$. In conclusione abbiamo che il prodotto di tutti gli elementi non nulli della prima sequenza deve essere congruo modulo p al prodotto degli elementi non nulli della seconda sequenza, cioè

$$a^{p-1}(p-1)! \equiv (p-1)! \bmod p.$$

Quindi $p|(a^{p-1}-1)(p-1)!$. Ma p non divide $(p-1)!$ e, di conseguenza, si ottiene $p|(a^{p-1}-1)$. □

Nota Bene 1.3. *Moltiplicando per a la tesi della forma forte del Piccolo Teorema di Fermat si ottiene la congruenza* $a^p \equiv a \bmod p$ *(ossia la forma debole) che è valida, come già sappiamo, anche per gli* $a \equiv 0 \bmod p$.

1.5.2 Classi residuali primitive

Nel caso in cui n non sia primo, in $\mathbb{Z}_n$ esistono alcuni elementi che non ammettono inverso moltiplicativo (e quindi $\mathbb{Z}_n$ non è un campo). Si considerino, per esempio, gli elementi 3 e 5 in $\mathbb{Z}_{15}$. Sia 3 che 5 non hanno inverso moltiplicativo perché $3 \cdot 5 = 15 \equiv 0 \bmod 15$. Dovremo allora limitarci a quegli elementi che possiedono inverso.

Definizione 1.4. *Diremo classi residuali primitive modulo n gli elementi di* $\mathbb{Z}_n^*$. *Denoteremo con* $\varphi(n)$ *(*funzione di Eulero*) il numero di elementi di* $\mathbb{Z}_n^*$.

Studiamo ora alcune proprietà della funzione $\varphi(n)$.

Sia $n = p^\alpha$. Allora $(a,n) > 1 \Leftrightarrow p|a \Leftrightarrow a$ è uno dei $p^{\alpha-1}$ multipli di p tra 1 e p^α. In pratica $a \in \{p, 2p, 3p, \ldots, p^{\alpha-1}p\}$. I rimanenti $p^\alpha - p^{\alpha-1}$ elementi più piccoli di p^α rappresentano differenti classi residuali primitive modulo p^α e quindi

$$\varphi(p^\alpha) = p^{\alpha-1}(p-1).$$

In particolare per $\alpha = 1$ si ha $\varphi(p) = p - 1$. Consideriamo adesso $n = \prod_i p_i^{\alpha_i}$. Ci servirà la seguente

Proposizione 1.6. *La funzione* φ *di Eulero è moltiplicativa, cioè*

$$\varphi(mn) = \varphi(m)\varphi(n) \quad \text{per } (m,n) = 1.$$

Dim. Sia $j \in \{0, \ldots, mn-1\}$ e siano $j_1 \equiv j \bmod m$, $j_2 \equiv j \bmod n$. Per il Teorema Cinese del Resto 1.5, per ogni coppia j_1, j_2 esiste un unico $j \in \{0, \ldots, mn-1\}$ tale che $j_1 \equiv j \bmod m$, $j_2 \equiv j \bmod n$. Si noti inoltre che $(j,mn) = 1$ se e solo se $(j,m) = 1 = (j,n)$ e che quest'ultima condizione equivale a $(j_1,m) = 1$ e $(j_2,n) = 1$. Quindi l'insieme $\{j \in \{0, \ldots, mn-1\} : (j,mn) = 1\}$ è in corrispondenza biunivoca con l'insieme $\{(j_1,j_2) : j_1 \in \{0, \ldots, m-1\}, j_2 \in \{0, \ldots, n-1\} \text{ e } (j_1,m) = 1 = (j_2,n)\}$. Da ciò segue che i due insiemi hanno lo stesso numero di elementi ed allora $\varphi(mn) = \varphi(m)\varphi(n)$. □

Osservazione 1.6. *Se si è dimostrata nell'Osservazione 1.4, nel caso* $(m,n) = 1$, *la biiezione tra* $\mathbb{Z}_{mn}^*$ *e* $\mathbb{Z}_m^* \times \mathbb{Z}_n^*$, *la Proposizione 1.6 segue immediatamente da essa essendo, per definizione,* $\varphi(l) = \text{card}(\mathbb{Z}_l^*)$.

Grazie alla Proposizione 1.6 possiamo finalmente scrivere quanto vale la funzione di Eulero nel caso generale. Se $n = \prod_i p_i^{\alpha_i}$, p_i distinti, abbiamo che

$$\varphi(n) = \prod_i p_i^{\alpha_i - 1}(p_i - 1) = n \prod_i \left(1 - \frac{1}{p_i}\right).$$

La seguente Proposizione esamina la complessità computazionale del calcolo di $\varphi(n)$ in relazione alla conoscenza della fattorizzazione di n. Ci servirà quando dovremo esaminare la sicurezza del sistema RSA.

Proposizione 1.7. *Sia $n = pq$ con $p \neq q$ primi. Allora possiamo calcolare $\varphi(n)$ a partire da p, q con complessità computazionale $O(\log n)$ e possiamo calcolare p, q a partire da $\varphi(n)$ con complessità computazionale $O\left(\log^3 n\right)$.*

Dim. Se n è pari allora possiamo dire che $p = 2$ e quindi $q = \frac{n}{2}$. Allora $\varphi(n) = \frac{n}{2} - 1$. Se n è dispari segue che

$$\varphi(n) = (p-1)(q-1) = n + 1 - (p+q).$$

Allora a partire da p, q e n si ottiene $\varphi(n)$ con una addizione e una sottrazione (complessità $O(\log n)$). D'altra parte, conoscendo $\varphi(n)$ e n, otteniamo $p + q = n + 1 - \varphi(n)$. Quindi di p, q conosciamo la somma ed il prodotto; da ciò deduciamo che p, q sono le soluzioni dell'equazione $x^2 - (p+q)x + n = 0$. In definitiva, posto $2b = p + q$, abbiamo $p, q = b \pm \sqrt{b^2 - n}$. Il costo computazionale della estrazione della radice quadrata è $O\left(\log^3 n\right)$ (si utilizzi sull'espansione binaria di n il ben noto algoritmo studiato nella scuola media, oppure il Metodo di Newton). □

Presentiamo adesso il seguente risultato che fa uso della funzione φ di Eulero e che ci sarà utile nella decifratura del sistema RSA.

Proposizione 1.8. *Sia n privo di quadrati e d, e tali che $\varphi(n) | (de - 1)$. Allora*

$$a^{de} \equiv a \bmod n \quad \text{per ogni } a \in \mathbb{Z}_n.$$

Dim. Il fatto che n sia privo di quadrati significa che $n = \prod_i p_i$ e $p_i \neq p_j$ per $i \neq j$. Allora l'ipotesi $\varphi(n) | (de - 1)$ equivale a $(p_i - 1) | (de - 1)$ per ogni $p_i | n$. Quindi, per la forma forte del Piccolo Teorema di Fermat 1.7, otteniamo

$$a^{de-1} = a^{(p_i-1)k_i} = (a^{(p_i-1)})^{k_i} \equiv 1^{k_i} \equiv 1 \bmod p_i$$

per ogni $p_i | n$, $a \in \mathbb{Z}_n^*$. Moltiplicando per a entrambi i lati della congruenza precedente si ottiene che $a^{de} \equiv a \bmod p_i$ per ogni $p_i | n$, $a \in \mathbb{Z}_n^*$. Inoltre è chiaro che, se $p_i | a$, si ha che $a^{de} \equiv a \bmod p_i$. Possiamo quindi concludere che $a^{de} \equiv a \bmod p_i$

per ogni $p_i|n$, $a \in \mathbb{Z}_n$. Applicando il Teorema Cinese del Resto 1.5, segue che esiste una unica soluzione modulo n del sistema di congruenze $x \equiv a \bmod p_i$ per ogni $p_i|n$. Otteniamo allora che a^{de} è l'unica soluzione cercata e quindi $a^{de} \equiv a \bmod n$ per ogni $a \in \mathbb{Z}_n$. □

Poiché nel sistema RSA utilizzeremo solo interi privi di quadrati, potremo utilizzare la Proposizione precedente. Nel caso n sia di forma generale, abbiamo la seguente generalizzazione del Piccolo Teorema di Fermat 1.7.

Teorema 1.8 (Eulero-Fermat). *Sia $n, a \in \mathbb{N}$ e $(a,n) = 1$. Allora*

$$a^{\varphi(n)} \equiv 1 \bmod n.$$

Dim. Sia $n = p^\alpha$. Procediamo per induzione su α:

- passo 1): $\alpha = 1$. Quindi $\varphi(p) = p-1$ e allora la tesi vale per il Piccolo Teorema di Fermat 1.7 nella forma forte;

- passo 2): supponiamo che la tesi sia verificata per $\alpha - 1$; la proviamo per α. Per ipotesi induttiva abbiamo che esiste $b \in \mathbb{Z}$ tale che

$$a^{p^{\alpha-1}-p^{\alpha-2}} = 1 + p^{\alpha-1}b.$$

Eleviamo alla p entrambi i lati dell'equazione precedente. Otteniamo, per il Binomio di Newton, che

$$(a^{p^{\alpha-1}-p^{\alpha-2}})^p = (1+p^{\alpha-1}b)^p = \sum_{k=0}^{p} \binom{p}{k} (p^{\alpha-1}b)^k.$$

Ma tutti i coefficienti binomiali $\binom{p}{k}$, $k \neq 0, p$, sono divisibili per p e quindi possiamo scrivere che

$$\sum_{k=0}^{p} \binom{p}{k} (p^{\alpha-1}b)^k = 1 + p^\alpha \sum_{k=1}^{p} a_k,$$

dove gli a_k sono degli interi opportuni. In definitiva abbiamo quindi che $a^{p^\alpha - p^{\alpha-1}} \equiv 1 \bmod p^\alpha$, cioè

$$a^{\varphi(p^\alpha)} \equiv 1 \bmod p^\alpha.$$

Per il metodo di induzione abbiamo quindi che la tesi vale per tutte le potenze prime. Poniamo adesso $n = p^\alpha q^\beta$, $p \neq q$ primi, α, β interi positivi. Per quanto visto in precedenza sappiamo che $a^{\varphi(p^\alpha)} \equiv 1 \bmod p^\alpha$ e che $a^{\varphi(q^\beta)} \equiv 1 \bmod q^\beta$. Elevando la prima alla $\varphi(q^\beta)$ e la seconda alla $\varphi(p^\alpha)$ otteniamo il sistema

$$\begin{cases} (a^{\varphi(p^\alpha)})^{\varphi(q^\beta)} \equiv 1 \bmod p^\alpha \\ (a^{\varphi(q^\beta)})^{\varphi(p^\alpha)} \equiv 1 \bmod q^\beta \end{cases}$$

che, per il Teorema Cinese del Resto 1.5, ammette un'unica soluzione modulo $p^\alpha q^\beta$. Quindi

$$a^{\varphi(p^\alpha)\varphi(q^\beta)} \equiv 1 \bmod p^\alpha q^\beta$$

è l'unica soluzione cercata ed il teorema segue per la moltiplicatività di φ. Il caso in cui n abbia forma generale si risolve applicando il Teorema Fondamentale dell'Aritmetica 1.1, la moltiplicatività di φ (Proposizione 1.6) e quanto visto sopra. □

Osserviamo che per calcolare esplicitamente l'inverso di una classe residuale possiamo sfruttare il fatto che $b^{-1} \equiv b^{\varphi(n)-1} \bmod n$. Infatti, visto che $b^{\varphi(n)} \equiv 1 \bmod n$ per il Teorema di Eulero-Fermat 1.8, è sufficiente moltiplicare entrambi i lati per l'inverso di b. Osserviamo però che tale metodo è computazionalmente non buono perché richiede il calcolo di $\varphi(n)-1$ che è piuttosto oneroso (a meno di non conoscere a priori la fattorizzazione completa di n). Per il calcolo degli inversi è preferibile utilizzare l'Algoritmo Euclideo e la Formula di Bézout 1.2 che forniscono un metodo estremamente più efficiente.

Un altro interessante risultato è la seguente

Proposizione 1.9. *Sia $n, a \in \mathbb{N}$, $(a,n) = 1$ e $l \equiv l' \bmod \varphi(n)$. Allora*

$$a^l \equiv a^{l'} \bmod n.$$

Dim. Abbiamo che $l = l' + k\varphi(n)$ per un certo $k \in \mathbb{Z}$. Allora, moltiplicando $a^{\varphi(n)} \equiv 1 \bmod n$ per se stessa k volte, otteniamo $a^{k\varphi(n)} = a^{l-l'} \equiv 1 \bmod n$. La tesi segue moltiplicando entrambi i membri per $a^{l'}$. □

1.6 Crittografia

1.6.1 Terminologia e notazioni

Con il termine *Crittografia* intendiamo lo studio di metodi atti ad inviare messaggi in forma "nascosta" in modo tale che solamente il destinatario possa rimuovere l'impedimento e leggere il messaggio in forma "chiara". Denoteremo i due insiemi fondamentali con

$$\mathcal{P} = \{\text{messaggi in forma "chiara"}\}$$

e

$$\mathcal{C} = \{\text{messaggi in forma codificata}\}.$$

Per *Trasformazione Crittografica* intenderemo una funzione f *iniettiva* da $\mathcal{P}$ a $\mathcal{C}$

$$\mathcal{P} \xrightarrow{\quad f \quad} f(\mathcal{P}) \xrightarrow{\quad f^{-1} \quad} \mathcal{P}.$$

La funzione f deve essere iniettiva perché vogliamo eliminare ogni tipo di ambiguità nella decifrazione. Diremo anche che f è la funzione di cifratura e che f^{-1} è la funzione di decifratura. Usualmente sia la funzione di cifratura che quella di decifratura sono dipendenti da un parametro. In questo ambito tali parametri vengono rispettivamente denominati:

1. *Chiave di Cifratura (Enciphering Key)*: K_E;
2. *Chiave di Decifratura (Deciphering Key)*: K_D.

Il ruolo di queste due chiavi (ad esempio le relazioni tra loro intercorrenti e la caratteristica di doverle mantenere entrambe segrete o di poterne rendere una pubblica) consente di classificare i metodi crittografici.

Nel seguito chiameremo *Crittosistema* una quaterna $(\mathcal{P}, \mathcal{C}, f, f^{-1})$.

1.7 Cenni storici

Uno dei primi esempi conosciuti di utilizzo di metodi crittografici risale a *Giulio Cesare*. In termini moderni possiamo dire che i Romani utilizzavano un metodo basato sulle proprietà dell'aritmetica modulo n (n indica il numero di lettere dell'alfabeto utilizzato). In pratica si poneva in corrispondenza biunivoca l'alfabeto con l'insieme $\mathbb{Z}_n$ e si cifravano i messaggi semplicemente "traslando" in avanti ogni lettera di una fissata quantità di posizioni m (ossia si addizionava ad m modulo n il corrispondente numerico di ogni lettera). Per esempio, se $n = 21$ e $m = 5$, per cifrare il messaggio P formato (per semplicità) di una sola lettera si utilizzava la funzione di cifratura data da

$$C \equiv P + 5 \bmod 21$$

e di decifratura data da

$$P \equiv C - 5 \bmod 21.$$

È chiaro che, con un poco di pazienza ed utilizzando un'analisi di frequenza delle lettere, si può facilmente *violare* il crittosistema descritto (ossia si può leggere P a partire da C ricavandosi autonomamente la funzione di decifratura).

Si possono facilmente pensare alcune varianti del crittosistema precedente: per esempio, se si mandano messaggi di due lettere utilizzando un alfabeto di 22 lettere (21 lettere più lo spazio bianco), possiamo far corrispondere a $P = \alpha\beta$ il numero

$$22x + y \in \{0, \dots, 483\} \quad x, y \in \{0, \dots, 21\},$$

dove x, y sono i rispettivi corrispondenti numerici dei simboli α, β. In tal modo intendiamo ogni lettera come una cifra in base 22 ed ogni messaggio di due lettere viene cifrato con un numero di due cifre in base 22.

Chiaramente il ragionamento si può estendere al caso di messaggi formati da k lettere prese da un alfabeto di n simboli (allora i messaggi vengono cifrati come interi compresi tra 0 e n^k-1).

Nel 16° secolo, utilizzando ancora delle idee basate essenzialmente sulle congruenze di interi, *Vigenère* inventò una variante del sistema precedente che è meno semplice da violare. In pratica venivano cifrati blocchi di k lettere (un vettore di $(\mathbb{Z}_n)^k$) traslando ogni blocco di una "parola chiave" di k lettere (addizionando cioè a $P \in (\mathbb{Z}_n)^k$ un certo vettore fissato $B \in (\mathbb{Z}_n)^k$).

Come nel caso precedente il sistema può essere violato analizzando le frequenze di occorrenza in ogni progressione aritmetica modulo k dei corrispondenti numerici delle lettere. Inoltre in tal modo si può anche risalire alla "parola chiave" B.

Nel 1931 *Hill* utilizzò come funzione f il prodotto righe per colonne con una matrice invertibile. Ossia posto $\mathcal{P} = \mathcal{C} = (\mathbb{Z}_n)^k$ si considera una matrice A invertibile ad elementi in $\mathbb{Z}_n$ (cioè $(\det A, n) = 1$). Allora la cifratura si ottiene mediante prodotto righe per colonne di A con il vettore $P \in \mathcal{P}$. La decifratura avviene ovviamente mediante prodotto righe per colonne di $C \in \mathcal{C}$ con $A^{-1} \bmod n$.

La violazione di tale metodo può essere ottenuta utilizzando l'algebra lineare modulo n nel caso si sia riusciti a conoscere alcune coppie P e C (si veda, ad esempio, il §5.6.3 di [6] oppure [7]).

Come si può notare dagli esempi precedenti, nella maggior parte dei crittosistemi veniva fatto uso solamente di argomenti elementari di Algebra e Teoria dei Numeri. Tale situazione si è protratta fino alla metà degli anni settanta del secolo scorso.

1.8 Crittografia classica

Gli esempi precedentemente menzionati riguardano tutti esempi di *crittografia classica*. Tali metodi appartengono alla categoria detta a *chiave privata* (ponendo l'enfasi sul fatto che K_E e K_D devono essere tenute entrambe segrete). Se si considerano le relazioni tra le chiavi sono anche detti a *chiave simmetrica* perché, come vedremo, K_E e K_D hanno un ruolo simmetrico dato che da una si può ottenere l'altra con basso costo computazionale.

Ciò significa che chi ha abbastanza informazioni per codificare il messaggio ha *automaticamente* abbastanza informazioni per decifrare. Quindi ogni coppia di utenti che vuole comunicare segretamente, deve essere riuscita a scambiarsi le chiavi di cifratura e decifratura in un modo *sicuro* (per esempio mediante un corriere di fiducia). D'altra parte, se esiste una comunicazione sicura, perché preoccuparsi di instaurare un nuovo metodo di comunicazione che potrebbe essere non sicuro? Ciò avviene perché esiste sempre la possibilità che il corriere non sia effettivamente degno di *totale fiducia* o che il messaggio a lui affidato venga intercettato e fini-

sca a persone che invece vorremmo mantenere all'oscuro del contenuto delle nostre comunicazioni (usando altre parole: la sicurezza assoluta non esiste e nessuna precauzione è superflua).

Negli esempi precedenti le chiavi dei cifratura e decifratura erano:

Metodo	cifratura K_E	decifratura K_D
Cesare	m	$-m$
Vigenère	vettore v	$-v$
Hill	matrice A	$A^{-1} \bmod n$.

Facciamo anche notare che i metodi classici hanno anche la caratteristica che, dal punto di vista della *complessità computazionale*, la *cifratura e la decifratura sono equivalenti* (ossia il loro calcolo richiede una complessità computazionale identica o dello stesso ordine). Da tale osservazione possiamo fare discendere la seguente definizione basata proprio sulla complessità computazionale.

Definizione 1.5 (Crittografia classica). *Crittosistemi in cui, una volta che il messaggio cifrato e K_E siano noti, K_D e la decifratura di tale messaggio possono essere calcolate in ordine di tempo approssimativamente uguale a quello necessario per cifrare.*

In altri termini, noto K_E, la complessità computazionale necessaria per determinare K_D e per eseguire la decifratura di un messaggio codificato è dello stesso ordine di grandezza della complessità della cifratura.

Vediamo adesso qualche esempio di metodi classici.

Esempio 1.5. *Il metodo della* traslazione *di* Giulio Cesare *con un alfabeto di N lettere e traslazione di $1 \leq l < N$ posizioni si compone dei seguenti passi:*

(1) si traduce una lettera del messaggio in intero m modulo N, $m \in \{0,1,2,\ldots,N-1\}$;

(2) la cifratura *avviene sommando modulo N all'equivalente numerico della lettera la traslazione; ossia $m \to c = m+l \bmod N$;*

(3) la decifratura *avviene sottraendo modulo N al messaggio codificato c la traslazione; ossia $c \to m = c-l \bmod N$.*

Schematicamente applicando quanto sopra alla parola CIAO si ottiene:
CIAO $\to$ 2 8 0 12 $\to$ 5 11 3 15 (traslazione di 3) $\to$ FNCR $\to \ldots \to$ CIAO

Esempio 1.6. *Usare il metodo di Vigenère per codificare il messaggio*
QUESTO METODO CRITTOGRAFICO NON E' SICURO
usando l'alfabeto

"A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z", ",", ".", " ' ", " "

e la parola chiave FREQUENZA. Utilizzate come equivalenti numerici delle lettere dell'alfabeto gli interi appartenenti all'insieme $\{1,\dots,30\}$. *Verificare che si ottiene il crittogramma*

WIJFKTNIFZCIBUHBEUZCLEVKW'PFBTAUJMZTOUZEF

e la correttezza dell'operazione di decifratura.

Esempio 1.7. *Riportiamo qui un famoso esempio di crittogramma tratto dal racconto "Lo scarabeo d'oro" di E.A. Poe. In esso è descritta la ricerca di un tesoro a partire da un pezzo di pergamena che contiene il disegno di un capretto e una serie di caratteri, come quelli qui riprodotti.*

5 3 ‡ ‡ † 3 0 5)) 6 * ; 4 8 2 6) 4 ‡
.) 4 ‡) ; 8 0 6 * ; 4 8 † 8 ¶ 6 0))
8 5 ; 1 ‡ (; : ‡ * 8 † 8 3 (8 8) 5 *
† ; 4 6 (; 8 8 * 9 6 * ? ; 8) * ‡ (;
4 8 5) ; 5 * † 2 : * ‡ (; 4 9 5 6 * 2
(5 * – 4) 8 ¶ 8 * ; 4 0 6 9 2 8 5) ;
) 6 † 8) 4 ‡ ‡ ; 1 (‡ 9 ; 4 8 0 8 1 ;
8 : 8 ‡ 1 ; 4 8 † 8 5 ; 4) 4 8 5 † 5 2
8 8 0 6 * 8 1 (‡ 9 ; 4 8 ; (8 8 ; 4 (
‡ ? 3 4 ; 4 8) 4 ‡ ; 1 6 1 ; : 1 8 8 ;
‡ ? ;

Nel vedere questo schema, ci sono di solito due reazioni: la prima, che possiamo descrivere con le parole di Amleto

> HAMLET*: "O dear Ophelia, I am ill at these numbers"*
> *William Shakespeare, "Hamlet," II, 2; 119*

si può definire come rassegnazione di fronte a un problema apparentemente insolubile, mentre la seconda, per la quale prendiamo in prestito le parole di Alice,

> ALICE*: "Somehow, it seems to fill my head with ideas—only I don't know exactly what they are!"*
> *Lewis Carroll, "Through the Looking Glass"*

si può definire possibilista. Come possiamo aiutare Amleto ed Alice a scoprire il significato dei simboli del crittogramma? Seguiremo abbastanza da vicino la descrizione che l'Autore stesso fa della decifrazione.

La prima cosa da fare è un'ipotesi sulla lingua del testo del crittogramma. L'Autore spiega che il manoscritto è stato rinvenuto in una zona un tempo infestata da pirati, e che uno dei più famosi pirati si chiamava Kidd: poiché in inglese "kid" vuol dire capretto, in mancanza di altre informazioni la prima ipotesi sulla lingua del crittogramma è che questa sia l'inglese.

L'ANALISI DI FREQUENZA

Il secondo passo da fare si chiama analisi di frequenza, ed in effetti prescinde dalla conoscenza della lingua in cui è scritto il testo originale: contiamo quante volte compaiono i singoli caratteri.

8	33	*	13	1	8	3	4
;	26	5	12	0	6	?	3
4	19	6	11	2	5	¶	2
‡	16	(	10	9	5	.	1
)	16	†	8	:	4	–	1

È un'osservazione piuttosto banale che in ogni lingua alcune lettere sono molto più frequenti di altre: in particolare, le vocali tendono a comparire più spesso delle consonanti. Se la nostra congettura sulla lingua del testo originale è corretta, i dati in questa tabella suggeriscono che il simbolo "8" rappresenti con ogni probabilità la lettera "e" dato che in un normale testo inglese questa lettera da sola compare circa il 13% delle volte. Quando usiamo la parola "normale" intendiamo dire che il testo non *è stato costruito con l'intento di distorcere appositamente la normale frequenza delle lettere.*

Il fatto che il simbolo più frequente "8" compaia 33 volte su 203 caratteri (con una frequenza relativa del 16% circa) suggerisce anche che lo spazio *fra le parole è stato probabilmente eliminato, perché in caso contrario sarebbe il carattere di gran lunga più frequente.*

Non potendo sapere dove iniziano e finiscono le parole (cosa che darebbe una miniera di informazioni in lingue come l'italiano nelle quali le lettere finali sono quasi esclusivamente vocali) e neppure quali sono le sillabe più frequenti, facciamo un altro passo di analisi statistica, basato sull'osservazione che il comportamento delle vocali e delle consonanti è radicalmente diverso: infatti, le vocali hanno la proprietà di poter precedere o seguire qualunque altra lettera dell'alfabeto (con qualche eccezione relativamente infrequente, come nel caso della consonante "q" che è sempre seguita dalla vocale "u" o dalla "q" stessa), mentre le consonanti tendono a precedere o seguire un numero ristretto di altre lettere. In altre parole, la maggior parte di combinazioni consonante-consonante dà luogo a sequenze

impronunciabili, mentre ciò non è vero per le combinazioni di vocali e consonanti.

Facciamo dunque una nuova analisi di frequenza, contando questa volta il numero delle occorrenze dei digrafi*, cioè delle coppie di lettere adiacenti: nella tabella seguente abbiamo raccolto i risultati di questo conteggio, trascurando naturalmente i digrafi meno frequenti, cioè quelli che compaiono meno di 4 volte.*

;	4	12	8	5	5	†	8	4
4	8	8	8	8	5	(	;	4
6	∗	5	4	‡	4	8	)	4
)	4	5	;	8	4			

Il risultato conferma la nostra ipotesi a proposito del simbolo "8" che compare in compagnia di molti simboli diversi, precedendoli o seguendoli, e che spesso è raddoppiato: come si sa, il dittongo "ee" è piuttosto frequente nella lingua inglese.

Prima di sostituire il simbolo "8" con la lettera "e" spingiamo la nostra analisi statistica un passo avanti, esaminando anche i trigrafi*, cioè terne di lettere adiacenti. Compiliamo dunque la tabella che segue, anche in questo caso ignorando i trigrafi meno frequenti.*

;	4	8	7	∗	;	4	3
)	4	‡	4	8	†	8	3
				‡	(	;	3

Che cosa possiamo "dedurre" da questa tabella? Considerando il fatto che il simbolo "8" rappresenta probabilmente la lettera "e" e che una delle parole più frequenti della lingua inglese è l'articolo determinativo "the", è piuttosto ragionevole supporre che il trigrafo più frequente rappresenti per l'appunto proprio questa combinazione di lettere. Non si tratta di una vera e propria deduzione in senso matematico, ma di una ragionevole congettura. Siamo dunque pronti per la

PRIMA CONGETTURA: "; 48" = "THE"

Useremo la convenzione di scrivere in nero i simboli di cui non abbiamo ancora stabilito il valore, in blu le lettere sostituite ai simboli già determinati, ed in rosso le lettere su cui si concentra di volta in volta l'analisi di Poe. Il crittogramma originale diventa:

5	3	‡	‡	†	3	0	5	)	)	6	*	t	h	e	2	6	)	h	‡
.	)	h	‡	)	t	e	0	6	*	t	h	e	†	e	¶	6	0	)	)
e	5	t	1	‡	(	t	:	‡	*	e	†	e	3	(	e	e	)	5	*

† t h 6 (t e e * 9 6 * ? t e) * ‡ (t
h e 5) t 5 * † 2 : * ‡ (t h 9 5 6 * 2
(5 * – h) e ¶ e * t h 0 6 9 2 e 5) t
) 6 † e) h ‡ ‡ t 1 (‡ 9 t h e 0 e 1 t
e : e ‡ 1 t h e † e 5 t h) h e 5 † 5 2
e e 0 6 * e 1 (‡ 9 t h e t (e e t h (
‡ ? 3 h t h e) h ‡ t 1 6 1 t : 1 e e t
‡ ? t

Il narratore del racconto di Poe suggerisce di guardare la sequenza di lettere indicata in rosso: vediamo l'articolo "the" seguito da "t(eeth." L'ipotesi è che il simbolo "(" rappresenti la lettera "r" in modo che questa sequenza indichi la parola "tree" (che vuol dire albero) e sia poi seguita dall'inizio di un'altra parola. Se questa ipotesi, che è ragionevole anche in base alla frequenza relativa del simbolo corrispondente, è vicina alla verità, sostituendo la lettera "r" dovrebbero comparire altri frammenti di parole inglesi plausibili. Passiamo quindi alla

SECONDA CONGETTURA: "(" = "R"

Questo è il risultato della sostituzione indicata:

5 3 ‡ ‡ † 3 0 5)) 6 * t h e 2 6) h ‡
.) h ‡) t e 0 6 * t h e † e ¶ 6 0))
e 5 t 1 ‡ r t : ‡ * e † e 3 r e e) 5 *
† t h 6 r t e e * 9 6 * ? t e) * ‡ r t
h e 5) t 5 * † 2 : * ‡ r t h 9 5 6 * 2
r 5 * – h) e ¶ e * t h 0 6 9 2 e 5) t
) 6 † e) h ‡ ‡ t 1 r ‡ 9 t h e 0 e 1 t
e : e ‡ 1 t h e † e 5 t h) h e 5 † 5 2
e e 0 6 * e 1 r ‡ 9 t h e t r e e t h r
‡ ? 3 h t h e) h ‡ t 1 6 1 t : 1 e e t
‡ ? t

Si può forse essere d'accordo con un altro dei personaggi di "Amleto":

> POLONIUS*: "Though this be madness,*
> *yet there is method in't"*
> *William Shakespeare, "Hamlet," II, 2, 205–206*

Guardiamo ora il frammento di testo indicato in rosso: il narratore suggerisce che si tratti della parola "through" (attraverso), seguita dall'articolo determinativo

"the." Consultando la tabella delle frequenze dei vari simboli, scopriamo che "‡" compare ben 16 volte, mentre "?" e "3" compaiono rispettivamente 3 e 4 volte ciascuno. Questo è incoraggiante, perché effettivamente la vocale "o" è piuttosto frequente in inglese, mentre "u" e "g" sono lettere relativamente infrequenti. Siamo pronti per la nostra

TERZA CONGETTURA: "‡" = "O" "?" = "U" "3" = "G"

Come sempre, riportiamo il risultato della sostituzione dei simboli a cui abbiamo assegnato un, seppur ipotetico, valore.

5 g o o † g 0 5)) 6 * t h e 2 6) h o
.) h o) t e 0 6 * t h e † e ¶ 6 0))
e 5 t 1 o r t : o * e † e g r e e) 5 *
† t h 6 r t e e * 9 6 * u t e) * o r t
h e 5) t 5 * † 2 : * o r t h 9 5 6 * 2
r 5 * – h) e ¶ e * t h 0 6 9 2 e 5) t
) 6 † e) h o o t 1 r o 9 t h e 0 e 1 t
e : e o 1 t h e † e 5 t h) h e 5 † 5 2
e e 0 6 * e 1 r o 9 t h e t r e e t h r
o u g h t h e) h o t 1 6 1 t : 1 e e t
o u t

Per non tediare inutilmente i Lettori, condensiamo gli ultimi passi dell'analisi di Poe: per prima cosa concentriamoci sul primo frammento in rosso qui sopra. Compare quasi per intero la parola "degree" (grado) che possiamo considerare plausibile se la pergamena contiene davvero le indicazioni per trovare un tesoro. La seconda parola in rosso sembra essere "thirteen" (tredici): considerando che il simbolo "6" compare ben 11 volte e che dopo una parola come "grado" è naturale aspettarsi qualche dato di tipo numerico, ci sentiamo fiduciosi nel fare le congetture seguenti.

QUARTA CONGETTURA: "†" = "D" "6" = "I" "∗" = "N"

Continuando allo stesso modo, è possibile determinare il valore dei pochi simboli ancora sconosciuti: di seguito diamo la corrispondenza fra caratteri del testo cifrato e quelli del testo in chiaro.

8	e	*	n	1	f	3	g
;	t	5	a	0	l	?	u
4	h	6	i	2	b	¶	v
‡	o	(	r	9	m	.	p
)	s	†	d	:	y	–	c

Non resta che sostituire gli ultimi simboli con le lettere corrispondenti per ottenere il testo decifrato, o, per usare il gergo dei crittografi, "in chiaro."

IL MESSAGGIO IN CHIARO

a	g	o	o	d	g	l	a	s	s	i	n	t	h	e	b	i	s	h	o
p	s	h	o	s	t	e	l	i	n	t	h	e	d	e	v	i	l	s	s
e	a	t	f	o	r	t	y	o	n	e	d	e	g	r	e	e	s	a	n
d	t	h	i	r	t	e	e	n	m	i	n	u	t	e	s	n	o	r	t
h	e	a	s	t	a	n	d	b	y	n	o	r	t	h	m	a	i	n	b
r	a	n	c	h	s	e	v	e	n	t	h	l	i	m	b	e	a	s	t
s	i	d	e	s	h	o	o	t	f	r	o	m	t	h	e	l	e	f	t
e	y	e	o	f	t	h	e	d	e	a	t	h	s	h	e	a	d	a	b
e	e	l	i	n	e	f	r	o	m	t	h	e	t	r	e	e	t	h	r
o	u	g	h	t	h	e	s	h	o	t	f	i	f	t	y	f	e	e	t
o	u	t																	

Per comodità dei Lettori, riportiamo il messaggio decifrato dotandolo degli spazi e dei normali segni di interpunzione:

> *A good glass in the bishop's hostel in the devil's seat — forty-one degrees and thirteen minutes — northeast and by north — main branch seventh limb east side — shoot from the left eye of the death's-head — a bee-line from the tree through the shot fifty feet out.*

La sua traduzione in italiano è la seguente:

> *Un buon vetro nell'ostello del vescovo sulla sedia del diavolo — quarantun gradi e tredici minuti — nord-nordest — tronco principale settimo ramo lato est — cala dall'occhio sinistro del teschio — una linea retta dall'albero passando per il punto toccato lontano cinquanta piedi.*

Come si vede, si tratta della descrizione di una località dalla quale è possibile scorgere un albero, su un ramo del quale c'è un teschio. Muovendosi di cinquanta

m . r n l l s	e s r e u e l	s e e c J d e
s g t s s m f	u n t e i e f	n i e d r k e
k t , s a m n	a t r a t e S	S a o d r r n
e m t n a e I	n u a e c t	r r i l S a
A t v a a r	. n s c r c	i e a a b s
c c d r m i	e e u t u l	f r a n t u
d t , i a c	o s e i b o	K e d i i Y

Figura 1.6: Il crittogramma di Verne. Ci siamo presi la libertà di traslitterare il testo originario che usa l'alfabeto runico in quello latino, ed abbiamo sostituito il simbolo che sta per "mm" in questo alfabeto con "m."

piedi dall'albero nella direzione del punto che si trova sulla verticale dell'occhio sinistro del teschio, si troverà il luogo dove è stato seppellito il tesoro. Si noti che "vetro" è un'espressione gergale per cannocchiale.

C'è un aspetto poco verosimile nella descrizione, peraltro molto accurata, di Edgar Allan Poe, e cioè che il processo di decifrazione sia unidirezionale, un passo dietro l'altro, sempre più vicini alla soluzione corretta. In realtà è quasi certo che si commettano numerosi errori, che si finisca in vicoli ciechi, che si debba tornare sui propri passi, come in un labirinto: il filo di Arianna che guida il crittografo è rappresentato in buona parte dal suo intuito e dalla sua conoscenza delle caratteristiche della lingua in cui si suppone sia scritto il testo da decifrare, ma si basa su una solida analisi statistica del testo. Viceversa, un aspetto assolutamente verosimile riguarda la velocità con cui si decifra il messaggio una volta raggiunta una certa "massa critica" di caratteri identificati correttamente: i primi passi sono molto incerti e dubbi, mentre i passi successivi sono sempre più sicuri e rapidi.

Esempio 1.8. *Questo esempio riguarda un crittogramma tratto dal libro "Viaggio al centro della Terra" di J. Verne .*

Il luogo di accesso alla strada che conduce al centro della terra è nascosto in un crittogramma di natura completamente diversa da quello descritto nell'esempio precedente: lo riproduciamo nella Figura 1.6. Per la precisione, ci affrettiamo a ricordare che il crittogramma originale è ulteriormente protetto essendo scritto con l'alfabeto runico: per semplicità, qui abbiamo riprodotto la sua traslitterazione nell'alfabeto latino.

Questo crittogramma è del tipo "trasposizione": questo significa che le lettere

```
m . r n l l s
e s r e u e l
s e e c J d e
s g t s s m f
u n t e i e f
n i e d r k e
k t , s a m n
a t r a t e S
S a o d r r n
e m t n a e I
n u a e c t
r r i l S a
A t v a a r
. n s c r c
i e a a b s
c c d r m i
e e u t u l
f r a n t u
d t , i a c
o s e i b o
K e d i i Y
```

Figura 1.7: La trasposizione dei blocchi.

del testo in chiaro sono rimescolate (potremmo dire anagrammate) seguendo una certa regola, e cioè cambiate di posto rispetto alla loro sequenza originale, ma non sono cambiate in natura. In altre parole, le "a" rimangono "a," le "b" rimangono "b" e cosí via, ma la posizione delle lettere può essere cambiata. Questo fatto può essere scoperto dal crittanalista intento alla decifrazione per mezzo di una analisi di frequenza: infatti, non essendo stata cambiata la natura delle lettere che costituiscono il testo originale, la frequenza delle lettere del testo cifrato è la stessa della frequenza delle lettere del testo in chiaro, e quindi le lettere come le vocali compariranno con una frequenza prossima a quella attesa per qualche lingua.

Seguendo la trama del romanzo, il primo passo verso la decifrazione è una trasposizione dei blocchi che lo costituiscono: si veda la Figura 1.7. Il testo risultante viene poi letto in verticale:

```
messunkaSenrA.icefdoK.segnittamurtnecertserret
te,rotaivsadua,ednecsedsadnelacartniiiluJsira
tracSarbmutabiledmekmeretarcsilucoYsleffenSnI
```

e questo viene infine letto dal fondo:

```
InSneffelsYoculiscrateremkemdelibatumbra
ScartarisJuliiintracalendasdescende,audas
viator,etterrestrecentrumattinges.Kodfeci.
ArneSaknussem
```

IL TESTO IN CHIARO

Inserendo spazi e segni di interpunzione, possiamo finalmente ricostruire il testo originale:

> *In Sneffels Yoculis craterem kem delibat umbra Scartaris Julii intra calendas descende, audas viator, et terrestre centrum attinges. Kod feci. Arne Saknussem*

Si noti che il latino del testo lascia alquanto a desiderare: infatti, "kem" sta per "quem," "kod" per "quod," "audas" per "audax." La sua traduzione italiana è la seguente:

> *Nel cratere dello Yocul dello Sneffels che l'ombra dello Scartaris tocca alle calende di luglio discendi, audace viaggiatore, e raggiungerai il centro della terra. Ciò che io ho fatto. Arne Saknussem*

Metodi crittografici di questa natura sono usati molto di rado, perché è molto difficile accordarsi sul metodo di cifratura. In questo caso è stato utilizzato dall'autore per assicurarsi che un messaggio cosí delicato potesse essere decifrato solo da persone sufficientemente abili e determinate da poter intraprendere con successo il viaggio verso il centro della terra.

Esempio 1.9. *Nei primi anni del ventesimo secolo presero piede delle* macchine cifranti. *Tra esse una delle più famose, a causa del fatto che la sua violazione consentì agli Inglesi di conoscere in anticipo molte informazioni riservate tedesche durante la Seconda Guerra Mondiale, è la macchina di cifratura* Enigma. *Essa fu sviluppata nel 1920 dall'imprenditore tedesco* Scherbius. *Era una macchina elettromeccanica a rulli (da 3 a 8 a seconda delle versioni) e funzionava mediante una ingegnosa* iterazione *del metodo della traslazione di Cesare. Per questa macchina l'analisi di frequenza non bastava per riuscire a violare il sistema. Una prima violazione parziale di Enigma fu dapprima fatta da* M. Rejewski *e successivamente essa fu definitivamente raggiunta dagli Inglesi (per la precisione dal gruppo guidato da* A. Turing*). Lo studio dei calcolatori elettromeccanici e dei meccanismi teorici della computazione automatica necessari per violare Enigma portò Turing a progettare* Colossus*: un calcolatore elettromeccanico che al giorno d'oggi è considerato il primo computer della storia ed il vero progenitore degli attuali personal computer.*

Negli esempi precedenti possiamo notare che la chiave di cifratura e di decifratura hanno complessità computazionale *equivalente* (come richiesto dalla Definizione 1.5). Inoltre ottenere la chiave di decifratura da quella di cifratura è molto semplice; in alcuni casi è identica, in altri è ottenibile mediante una funzione facilmente calcolabile. Infine, quest'ultima caratteristica implica che la coppia di chiavi di cifratura/decifratura deve essere mantenuta segreta per impedire ad un intruso di inserirsi illegalmente nella comunicazione.

1.9 Crittografia a chiave pubblica (o asimmetrica)

Il concetto fondamentale necessario per costruire un metodo asimmetrico è che la complessità della decifratura deve essere, nel caso non si conosca K_D, di ordine superiore a quella della cifratura. Inoltre, per ottenere K_D a partire dalla sola conoscenza di K_E, va risolto un problema computazionalmente "difficile". D'altra parte è anche necessario che chi conosce K_D possa decifrare efficientemente (altrimenti anche il destinatario non potrebbe praticamente decifrare i messaggi a lui inviati ...).

La definizione di crittosistema asimmetrico è stata data nel 1976 da *Diffie* e da *Hellman* [4] e può essere espressa nel modo seguente:

Definizione 1.6. *Un* crittosistema asimmetrico *ha la proprietà che chi conosce solamente come cifrare* non può usare la chiave di cifratura K_E per risalire alla chiave di decifratura K_D *senza dover affrontare calcoli di lunghezza proibitiva.*

Osserviamo anche che, usualmente, le locuzioni *crittosistema asimmetrico* e *crittosistema a chiave pubblica* sono considerate sinonimi.

Infatti, la peculiarità dei metodi asimmetrici è che la chiave di cifratura K_E di ogni utente *viene resa pubblica* a tutti gli altri utenti perché essi la devono utilizzare per poter comunicare con il suo possessore. Ciò non inficia la sicurezza del sistema perché ottenere K_D a partire dalla sola conoscenza di K_E è praticamente impossibile.

Le funzioni che consentono di realizzare un metodo a chiave pubblica devono quindi avere la caratteristica che $f : \mathcal{P} \to \mathcal{C}$ è iniettiva e "facile" da calcolare una volta che K_E è nota ma, d'altra parte, deve essere molto "difficile" sia calcolare la funzione $f^{-1} : f(\mathcal{P}) \to \mathcal{P}$ senza avere informazioni addizionali (cioè senza conoscere la chiave di decifratura K_D) sia ottenere K_D a partire dalla sola conoscenza di K_E .

Funzioni che verificano tale proprietà vengono dette *funzioni unidirezionali* (o *one-way trap-door functions*).

Per essere più precisi, per poter costruire un sistema asimmetrico e poter quindi rendere *pubblica* K_E senza per questo esporre K_D, dobbiamo essere in presenza di una funzione f iniettiva il cui calcolo ha complessità è $O(\log^{c_1} B)$, dove B è il

parametro principale del sistema, e la cui funzione inversa ha, nel caso in cui K_D sia ignota, complessità $\geq c_2 \exp(c_3 \log^{\alpha} B)$ ($c_1, c_2, c_3 > 0$ e $\alpha \in (0,1]$ sono opportune costanti). Inoltre, per ottenere K_D a partire dalla sola conoscenza di K_E, deve essere necessario lo stesso tipo di complessità $\geq c_2 \exp(c_3 \log^{\alpha} B)$.

Ossia la funzione f deve avere complessità *polinomiale* nel numero di cifre di B mentre per *violare* il sistema è necessario risolvere un problema avente complessità almeno *sub-esponenziale* nel numero di cifre di B.

Allo stato attuale delle conoscenze non sono note funzioni che verificano questa condizione. Allora ci si accontenta, giocoforza, di funzioni che abbiano inversa per cui *si congettura* la calcolabilità con complessità $\geq c_2 \exp(c_3 \log^{\alpha} B)$ per "quasi tutti" i casi (questa è, ad esempio, la situazione relativa ai problemi della primalità e della fattorizzazione oppure del logaritmo discreto).

È comunque possibile, da un punto di vista teorico, che in futuro si possa verificare questa proprietà per una particolare trasformazione crittografica.

Nelle applicazioni, quindi, si lavora solamente con funzioni congetturalmente unidirezionali. Potrebbe accadere che la scoperta di nuovi algoritmi possa portare a invalidare tali congetture oppure potrebbe accadere che, a causa del miglioramento delle prestazioni dei microprocessori, funzioni che, per determinati parametri, sono oggigiorno considerate "praticamente" one-way tra dieci (o anche solo tre) anni non possano più essere ritenute tali.

Infine, anche a costo di risultare pedanti, puntualizziamo ancora una volta che *la peculiarità dei metodi a chiave pubblica (o asimmetrici)* è che *la chiave di cifratura va resa pubblica* per consentire la comunicazione mentre la chiave di decifratura *deve essere mantenuta segreta* (si ricordi che nei metodi classici vanno entrambe mantenute segrete).

Non avremo quindi più il problema di scambiare le chiavi; per comunicare con un utente sarà sufficiente ottenere la sua chiave pubblica da un pubblico elenco. Ciò consentirà anche di diminuire il numero totale di chiavi necessario: in un metodo classico con k utenti sono necessarie $\binom{k}{2}$ coppie di chiavi di cifratura/decifratura mentre in un metodo a chiave pubblica ne sono sufficienti k.

1.9.1 Il metodo del doppio lucchetto

Il fatto di eliminare lo *scambio delle chiavi* pare rendere intuitivamente impossibile la crittografia. In realtà, però, l'introduzione del concetto di funzione unidirezionale permette la costruzione di metodi che per funzionare non necessitano dello scambio delle chiavi.

Come primo esempio, vediamo come si può evitare lo scambio delle chiavi mediante il metodo del *doppio lucchetto*: supponiamo di avere due utenti A e B e che A voglia spedire un messaggio segreto a B:

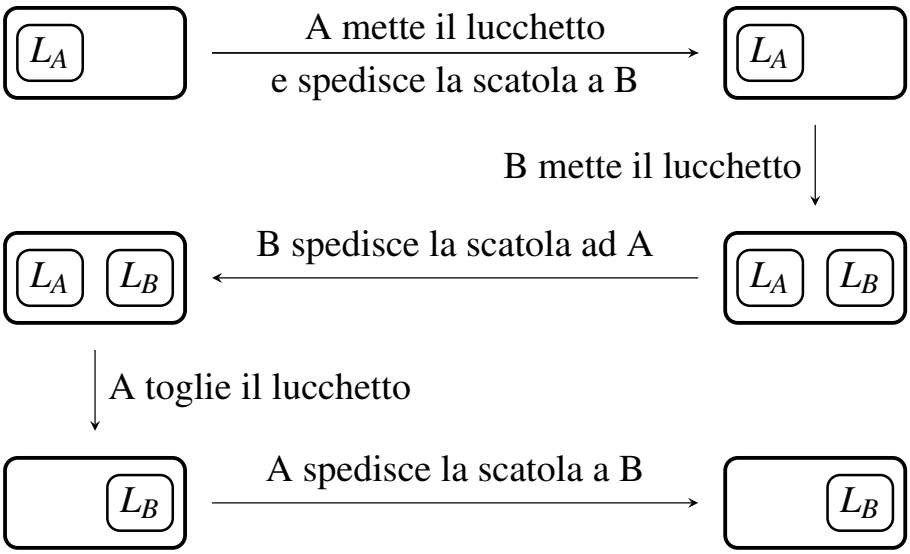

Figura 1.8: Schema di funzionamento del doppio lucchetto

(1) A mette il messaggio in una scatola che chiude con il suo lucchetto L_A (di cui lui solo ha la chiave) e che poi spedisce a B;

(2) B riceve la scatola chiusa con L_A, aggiunge il suo lucchetto L_B (di cui lui solo ha la chiave) e rispedisce il tutto ad A;

(3) A, ricevuta la scatola con il doppio lucchetto, toglie il lucchetto L_A e rispedisce la scatola a B;

(4) a questo punto, ricevuta la scatola, B può togliere il lucchetto L_B e leggere il messaggio di A.

La sicurezza di questo metodo risiede nel fatto che, supponendo che il lucchetto non sia forzabile, le chiavi per aprire i lucchetti sono conosciute e possedute solamente dai rispettivi proprietari (ed essi non le hanno dovute concordare e scambiare). In tal modo un terzo non può leggere né modificare il contenuto della scatola. Questo metodo viene applicato direttamente in alcuni crittosistemi e protocolli crittografici. Il suo principale problema è che la scatola viaggia più volte del necessario.

Passiamo ora ad introdurre l'esempio più famoso (ed anche più usato) di crittosistema a chiave pubblica.

1.10 Sistema RSA

Un esempio di funzione unidirezionale congetturale (ed anche il primo crittosistema a chiave pubblica ad essere ideato e sviluppato) è stato fornito nel 1978 da *Rivest*, *Shamir* e *Adleman* (si veda, per esempio, [6] o [7]). La loro idea fu quella di sfruttare la diversità di complessità computazionale presente tra il problema di riconoscere la primalità di un intero e quello di fornirne la fattorizzazione.

Descriviamo il suo funzionamento nelle linee generali:

(1) ogni utente sceglie in modo casuale due primi p, q distinti ed estremamente grandi (diciamo di circa 300 cifre decimali ciascuno) e pone $n = pq$;

(2) calcola $\varphi(n) = (p-1)(q-1) = \operatorname{card} \mathbb{Z}_n^*$;

(3) sceglie in modo casuale un intero e tale che $1 < e \le \varphi(n)$ e $(e, \varphi(n)) = 1$ (e coprimo con $\varphi(n)$);

(4) calcola $d \equiv e^{-1} \mod \varphi(n)$ (per lui è "facile" perché conoscendo sia p che q può calcolare facilmente $\varphi(n)$ e usare l'Algoritmo Euclideo per determinare d conoscendo e e $\varphi(n)$).

Osservazione 1.7. *In quanto sopra è importante notare che:*

(a) la scelta casuale *è fondamentale perché da essa dipende la sicurezza del crittosistema;*

(b) il calcolo di d al punto (4) di cui sopra viene ottenuto come sottoprodotto della verifica della proprietà $(e, \varphi(n)) = 1$ di cui al punto (2) (come abbiamo visto in precedenza).

Per *ogni utente* saranno quindi definite le seguenti quantità:

Chiave di Cifratura (pubblica): viene resa pubblica perché possa essere usata dagli altri utenti che vogliono comunicare con il possessore della chiave stessa. Essa è la coppia

$$K_E = (n, e).$$

Chiave di Decifratura (privata): d viene tenuta segreta perché con essa si realizza la decifratura dei messaggi pervenuti. Essa è

$$K_D = d.$$

Funzione di cifratura: è la funzione $f : \mathbb{Z}_n \to \mathbb{Z}_n$ data da

$$f(P) \equiv P^e \mod n.$$

Funzione di decifratura: è la funzione $f^{-1} : \mathbb{Z}_n \to \mathbb{Z}_n$ data da

$$f^{-1}(C) \equiv C^d \bmod n.$$

Per chiarire un poco il funzionamento del crittosistema, vediamo adesso l'esempio di un sistema con due soli utenti:

	Anna	Bruno
Pubblico	$K_{E_A} = (n_A, e_A)$	$K_{E_B} = (n_B, e_B)$
Privato	$K_{D_A} = d_A$	$K_{D_B} = d_B$.

Anna per mandare un messaggio P a Bruno dovrà utilizzare la chiave pubblica di Bruno e calcolare

$$C \equiv P^{e_B} \bmod n_B.$$

Bruno riceve C, applica la sua funzione di decifratura (segreta) e calcola

$$C^{d_B} \equiv P^{e_B d_B} \bmod n_B.$$

Ma $e_B d_B \equiv 1 \bmod \varphi(n)$. Quindi, per la Proposizione 1.8, si ha

$$C^{d_B} \equiv P \bmod n_B$$

ed allora Bruno può leggere il messaggio mandatogli da Anna.

Osservazione 1.8. *In pratica è* molto importante *che* $(P, n_B) = 1$. *Se così non fosse e si avesse che* $1 < u = (P, n_B) < n_B$, *allora si sarebbe determinato un fattore di* n_B *ed il sistema sarebbe stato violato da parte del mittente di* P.

Inoltre anche un intruso capace di intercettare i messaggi cifrati potrebbe sfruttare questo fatto. Infatti, se $1 < u = (P, n_B)$, *esisterebbero* $k, s, t \in \mathbb{Z}$ *tali che* $C = P^{e_B} + k n_B = (ut)^{e_B} + k(us)$ *e quindi si avrebbe che* $u \mid C$. *Pertanto* $1 < u | (C, n_B)$ *e quindi l'intruso potrebbe determinare un fattore non banale di* n_B *semplicemente calcolando* (C, n_B).

Da un punto di vista formale è possibile dunque definire, come abbiamo fatto in precedenza, la cifratura del metodo RSA su $\mathbb{Z}_n$, *ma per quanto sopra è meglio lavorare su* $\mathbb{Z}_n^*$.

Nel caso di un sistema con un numero di utilizzatori maggiore di due, sarà necessario creare ed inviare ad ogni utente un elenco delle chiavi pubbliche, in modo tale che ognuno possa trasmettere messaggi ad ogni altro utente del sistema stesso.

1.10.1 Sicurezza del sistema RSA

I punti fondamentali che assicurano la sicurezza del sistema sono:

(1) f è legata alla "facilità" computazionale di calcolare i primi p e q;

(2) *violare* il sistema (cioè calcolare f^{-1} senza conoscere K_D) è legato alla capacità di calcolare d conoscendo solo e e n. Per fare ciò bisogna essenzialmente essere capaci di calcolare $\varphi(n)$ conoscendo n (ma *senza* conoscere la sua fattorizzazione).

Ma nella Proposizione 1.7 abbiamo visto che:

- conoscendo p,q, il calcolo di $\varphi(n)$ costa $O(\log n)$ operazioni elementari;

- conoscendo $\varphi(n)$, il calcolo di p,q costa $O\left(\log^3 n\right)$ operazioni elementari;

ossia

conoscere $\varphi(n)$ è computazionalmente equivalente a conoscere la fattorizzazione di n.

Quindi la capacità di violare il sistema RSA dipende essenzialmente dall'essere in grado di trovare la fattorizzazione di n (che è *pubblico*) con complessità equivalente a quella con cui si prova la primalità di p,q. A parte casi particolari (ad esempio quelli in cui p,q sono molto piccoli o molto vicini a $\sqrt{n}$) sappiamo invece determinare la fattorizzazione di un intero solamente con un *alto* costo computazionale. È per tale motivo che RSA è considerato *sicuro*. Si veda anche il paragrafo seguente.

1.10.2 Cenni su algoritmi di primalità e fattorizzazione

Come abbiamo visto, la sicurezza del sistema RSA dipende dalla possibilità di dimostrare la primalità di un intero con una quantità di calcoli (e quindi di tempo) estremamente inferiore rispetto a quella necessaria per determinare almeno un fattore di un intero.

Per *algoritmo di primalità* intenderemo *un insieme finito di calcoli che (senza cercare i fattori di n) prova il fatto che n è primo*.

Alcuni degli algoritmi di primalità più conosciuti sono basati su proprietà delle congruenze e, sebbene i più semplici tra questi non siano in grado di concludere nulla di più che una valutazione della probabilità con cui n è primo, uno di essi (test di *Miller-Rabin*), assumendo la validità dell'Ipotesi Generalizzata di Riemann, prova la primalità di un intero n con complessità $O\left(\log^5 n\right)$.

Senza assumere alcuna ipotesi di tipo analitico ancora non dimostrata, il migliore algoritmo di primalità oggi noto è stato inventato nel 2002 da *Agrawal, Kayal* e *Saxena* [1] ed ha complessità computazionale (nella versione migliorata di Lenstra-Pomerance) pari a

$$O\big((\log n)^{6+\varepsilon}\big)$$

ossia "polinomiale" in $\log n$ (cioè polinomiale nel numero di cifre di n).

D'altra parte l'*algoritmo* banale di *fattorizzazione* (cioè un algoritmo che *fornisce almeno un fattore primo del numero n* in questione) è quello di provare a dividere l'intero n per tutti gli interi primi più piccoli della sua radice quadrata. Tale algoritmo, chiamato *Divisione per tentativi*, ha complessità computazionale $O(\sqrt{n}\log n)$. Una versione di questo metodo ed uno script PARI/Gp che lo realizza sono esposti nell'Incontro 3.4.

Sebbene molti studiosi abbiano cercato di affrontare il problema anche con tecniche estremamente sofisticate, purtroppo (o per fortuna ...) il problema della fattorizzazione in "tempo" polinomiale "resiste" ancora oggi.

Infatti neanche il migliore algoritmo di fattorizzazione oggi conosciuto (che è del 1993 ed è dovuto a Pollard-Lenstra-Lenstra, si veda, per esempio, [3] oppure [2]) ha complessità che avvicina quella degli algoritmi di primalità. Tale algoritmo, che non può essere esposto in questa sede perché è piuttosto sofisticato, è basato sulle proprietà di alcuni insiemi di numeri, detti *campi di numeri*, che generalizzano gli usuali insiemi numerici ed ha complessità computazionale che si *congettura* essere pari a

$$O\Big(e^{c\sqrt[3]{\log n(\log\log n)^2}}\Big), \text{ dove } c>0 \text{ è una costante fissata,}$$

cioè di tipo sub-esponenziale (ma che certamente non è polinomiale in $\log n$).

Come vedete, per i metodi di fattorizzazione si sanno fornire stime del caso peggiore, cioè delle maggiorazioni, mentre, per provare l'unidirezionalità della trasformazione crittografica di RSA, servirebbero opportune minorazioni.

Pertanto, poiché determinare minorazioni significative della complessità del caso medio della fattorizzazione è un problema aperto, il sistema RSA può essere considerato a chiave pubblica soltanto da un punto di vista congetturale.

1.10.3 Ordini di grandezza

Facciamo qualche piccolo calcolo per esemplificare quanto siano differenti gli ordini di grandezza di cui abbiamo parlato fino ad ora.

Per rendersi conto di quale sia la "velocità" di crescita della funzione esponenziale, si può osservare che, se pieghiamo a metà un foglio di carta un numero sufficientemente grande di volte, otteniamo uno spessore molto grande.

Per esempio, se il foglio ha uno spessore pari a $0,1$ mm, visto che con ogni piegatura raddoppiamo lo spessore totale, se fossimo in grado di piegarlo a metà per 42 *volte* otterremmo uno spessore totale maggiore della *distanza terra-luna*.

Infatti $2^{42} = 4398046511104$ e quindi lo spessore totale della carta piegata è di 4398046511104 decimillimetri ossia di 439804.6511104 km che è maggiore della distanza Terra-Luna (ricordiamo che essa è in media 384.000 km; 356.410 km al perigeo e 406.700 km all'apogeo).

Nelle applicazioni pratiche un personal computer oggigiorno in commercio (e che molti di voi avranno a casa) può calcolare numeri primi (p,q) di circa 300 cifre decimali ciascuno in *pochi secondi*. Il miglior algoritmo di fattorizzazione noto implementato su un moderno supercomputer per determinare i fattori di un intero n con 600 cifre decimali impiegherebbe *anni*.

A chi vuole avere più complete informazioni sugli algoritmi di primalità e fattorizzazione si consigliano i testi di Crandall-Pomerance [3] e di Cohen [2].

1.11 Firma digitale: schema generale

Una delle più importanti (e sorprendenti) applicazioni dei metodi a chiave pubblica è quella di consentire di *firmare* una sequenza di bit (un file, ad esempio).

In generale possiamo schematizzare la situazione come segue. A e B siano entrambi utenti di un sistema a chiave pubblica. Quindi f_A e f_B sono pubbliche e f_A^{-1} e f_B^{-1} sono segrete. Per questa applicazione assumiamo inoltre che f_A e f_B siano funzioni surgettive (oltre che iniettive).

A, per inviare un messaggio M a B, calcola $f_B(M)$.

Per *firmare* questo messaggio (ossia per consentire al destinatario di essere certo dell'identità del mittente) si può ragionare in questo modo:

(1) sia s_A un nome convenzionale di A (con numero progressivo, tempo di spedizione, numero IP macchina speditrice, ...);

(2) A certifica la propria identità a B inviandogli, oltre ad $f_B(M)$, anche la quantità $f_B(f_A^{-1}(s_A))$; quest'ultima quantità costituisce la *firma digitale* di A;

(3) B può decifrare il messaggio calcolando $f_B^{-1}(f_B(M)) = M$;

(4) B può poi verificare la firma di A (che può essere stata apposta solo da A stesso perché A è l'unico a conoscere f_A^{-1}) calcolando $f_A f_B^{-1}(f_B f_A^{-1}(s_A)) = s_A$.

Quali sono i problemi di questo schema? Il primo non è un problema crittografico ma generale: chi assicura la corrispondenza tra l'utente e la sua funzione di decifratura? Questo è lo stesso problema che si ha nel far corrispondere una persona fisica

ad un nome. Se non esiste qualcuno che certifica questa corrispondenza (rilasciando un documento di identità, per esempio) ognuno può dichiarare ciò che gli pare. Anche nella Crittografia si cerca di aggirare questo problema in modo analogo. Si inserisce nello schema una figura super-partes detta *Ente Certificatore* delle chiavi pubbliche. In tal modo B, dopo aver decifrato il messaggio e la firma di A, chiede all'Ente Certificatore di verificare la corrispondenza tra f_A e A (e spera che A non sia il padrone occulto dell'Ente Certificatore o non si sia preventivamente accordato con esso per frodarlo ...)

Il secondo problema dello schema generale è che un intruso C potrebbe intercettare la quantità $f_A^{-1}(s_A)$ oppure potrebbe identificare la sequenza di bit che la costituisce utilizzando un certo numero di messaggi intercettati. L'intruso C potrebbe poi fingersi A allegando la sequenza $f_A^{-1}(s_A)$ cosí com'è ad un suo messaggio in cui dichiara di essere A.

La contromisura usuale che viene adottata è quella di far dipendere la firma di un messaggio dal messaggio stesso.

1.11.1 Firma dipendente dal messaggio

Chiaramente si potrebbe utilizzare $s_A = M$ ma in tal modo si raddoppierebbe la lunghezza dell'informazione trasmessa ed il numero di calcoli necessari per la codifica e la decodifica. Di solito si preferisce supporre l'esistenza di una particolare funzione *pubblica* (detta *funzione hash*) che è nota a tutti gli utenti del crittosistema ed ha le proprietà seguenti:

(1) $h(M)$ (che viene detta detta *impronta* di M) è una sequenza di bit di lunghezza fissata (di solito è lunga almeno 160 bit);

(2) h non permette di risalire a M conoscendo solamente $h(M)$;

(3) la probabilità che $h(M) = h(M')$ con $M \neq M'$ è molto piccola (di solito la si vuole perlomeno minore di 10^{-50}).

La firma digitale viene allora calcolata da A come $f_A^{-1}(h(M))$ ed inviata a B con le usuali modalità.

Per verificare la firma, B decifra il messaggio e ottiene M; poi controlla che il mittente sia A e ottiene $h(M)$. Allora B ricalcola l'impronta di M (h è pubblica e B può calcolare $h(M)$ in modo autonomo) e controlla se la quantità pervenutagli e quella che si è calcolata autonomamente coincidono.

Questa aggiunta allo schema generale non risolve il problema della corrispondenza tra f_A ed A ma fornisce una firma che, oltre a non essere riutilizzabile per un

messaggio diverso (impedendo cosí ad un intruso che la intercetti di usarla in seguito), ha un ulteriore vantaggio perché certifica con buona affidabilità anche l'integrità del messaggio M.

1.11.2 Marcatura Temporale

L'introduzione dell'Ente Certificatore consente di eseguire anche una *marcatura temporale* (analoga all'autenticazione di un documento) del messaggio.

Supponiamo che A voglia inviare a B un messaggio e che voglia datarlo in modo che sia chiaro che lui, in data odierna, ha contattato B e che quest'ultimo non possa negare tale fatto. Supponiamo inoltre che l'Ente Certificatore disponga, come tutti gli altri utenti, di una funzione di cifratura f_E (pubblica) e di una funzione di decifratura f_E^{-1} (segreta). Inseriamo la marcatura temporale nello schema di firma con integrità del messaggio. Pertanto

(1) per datare un messaggio da spedire a B, A invia all'Ente Certificatore la quantità $f_B(f_A^{-1}(h(M)))$;

(2) l'Ente Certificatore aggiunge la data e l'ora T, applica la propria funzione segreta f_E^{-1} a quanto ha ricevuto ed invia $f_A(f_E^{-1}(f_B(f_A^{-1}(h(M))),T))$ (cioè il risultato) ad A dopo averlo codificato con f_A;

(3) A decodifica ed allega il risultato (che ha ricevuto da E) al messaggio M, codifica secondo le regole dei crittosistemi a chiave pubblica e spedisce il tutto a B;

(4) B riceve $f_B(M, f_E^{-1}(f_B(f_A^{-1}(h(M))),T))$ da A e decodificando ottiene M e la quantità $f_E^{-1}(f_B(f_A^{-1}(h(M))),T)$.

(5) per verificare l'integrità del messaggio, B utilizza f_E (che è pubblica) ed ottiene $(f_B(f_A^{-1}(h(M)),T)$ (cioè la firma di A che B può verificare come nel paragrafo precedente). Inoltre in tal modo B ha verificato la firma di E ed è quindi certo che la marcatura temporale T deve essere stato apposta da E perché nessun altro conosce f_E^{-1}.

Il punto fondamentale di questa situazione è che E è un utente "speciale" del crittosistema e si suppone che tutti gli altri utenti si fidino di lui.

CAPITOLO 2
PARI/GP: INTRODUZIONE

Questo capitolo è una rivisitazione (aggiornata alla versione 2.9.3 o superiore del software) del capitolo 9 di [6] e del testo dell'intervento tenuto dal primo autore al convegno OpenMath 2005, svoltosi a Vicenza il 14 dicembre 2005.

Verrà fornita una breve descrizione di alcune caratteristiche fondamentali del software preso in esame e si cercherà di esemplificarne l'uso. Molti altri esempi si troveranno anche nel Capitolo 3 all'interno dei singoli Incontri.

2.1 GENERALITÀ SU PARI/GP

PARI/GP è una combinazione di una libreria di aritmetica estesa (e molto altro ...) (`libpari`) e di un linguaggio di scripting (`gp`). È stato sviluppato da H. Cohen (Univ. Bordeaux 1) ed è attualmente mantenuto da K. Belabas (Univ. Bordeaux 1) con l'aiuto di alcuni volontari.

È uno strumento di calcolo utile in ambito professionale a chi si occupa di Teoria dei Numeri. Da un punto di vista didattico o amatoriale permette valutare l'efficienza "pratica" alcuni algoritmi crittografici senza dover preventivamente costruire autonomamente gli oggetti necessari (interi di lunghezza arbitraria, funzioni aritmetiche di base, ecc.).

In breve alcune sue caratteristiche fondamentali sono:

1) è un software free su licenza GNU, General Public License;

2) può essere integrato con interfacce grafiche Open-Source (X11);

3) può essere integrato in programmi WYSIWYG di editoria matematica (TeXMacs).

4) è multipiattaforma (Mac OS X, MS Windows, Linux, UniX);

5) dispone di una shell (`gp`) utilizzabile come calcolatore "avanzato". In tale modalità è anche disponibile un linguaggio di scripting (`GP`) la cui sintassi è analoga a quella del C e risulta quindi di facile utilizzo per la maggioranza degli utenti;

6) il programmi scritti nel linguaggio di scripting possono essere automaticamente tradotti in C con il software `gp2c` in modo da aumentarne la velocità di esecuzione;

7) le notazioni sono di tipo “usuale” per chi si occupa di matematica;

8) è sviluppato e costantemente aggiornato da specialisti del settore.

2.2 Primi passi con PARI/GP

2.2.1 Documentazione

La documentazione di PARI/GP si trova in internet all'indirizzo `http://pari.math.u-bordeaux.fr/` e contiene:

1) **Installation Guide:** guida all'installazione di PARI/GP su un computer UniX;

2) **Tutorial:** una guida stringata alle principali caratteristiche di PARI/GP;

3) **User's Guide:** il manuale vero e proprio;

4) **Reference Card:** un elenco delle principali funzioni.

Un secondo tutorial, molto più schematico del precedente, è quello di R. Ash e può essere trovato in rete all'indirizzo `http://www.math.uiuc.edu/~r-ash/GPTutorial.pdf`

2.2.2 Avviare una sessione di lavoro

Per avviare la sessione di PARI/GP è sufficiente dare il comando `gp` al prompt del sistema operativo. Appare allora un prompt del tipo `gp >`. Si otterranno delle risposte dalla shell; esse cominciano con il simbolo `%n`, dove n sta per un numero. Tale notazione indica il fatto che PARI/GP ha assegnato ad una variabile interna (identificata con % ed un numero progressivo) il valore indicato.

Vediamo un esempio di inizio di sessione:

```
 GP/PARI CALCULATOR Version 2.9.3 (released)
i386 running darwin (x86-64/GMP-6.1.2 kernel) 64-bit version
compiled: Oct  7 2017, Apple LLVM version 9.0.0 (clang-900.0.37)
threading engine: single
(readline v6.3 enabled, extended help enabled)
Copyright (C) 2000-2017 The PARI Group

PARI/GP is free software, covered by the GNU
General Public License, and
comes WITHOUT ANY WARRANTY WHATSOEVER.
```

```
Type ? for help, \q to quit.
Type ?15 for how to get moral (and possibly technical) support.

parisize = 8000000, primelimit = 500000
```

Ciò significa che al momento del caricamento del programma, PARI/GP alloca della memoria ed esegue delle precalcolazioni indicate dalle variabili `parisize = 8000000`, `primelimit = 500000`. Esse indicano la quantità di memoria allocata (in byte) per il funzionamento di PARI/GP (`parisize`) e che sono stati precalcolati tutti i primi minori o uguali a `primelimit` (tale valore influenza i valori utilizzati da molte delle funzioni aritmetiche che vedremo in seguito (ed anche da altre: `forprime`, per esempio)).

Se tali valori non sono sufficienti si può procedere come segue:

1) eseguire `gp` mediante il comando `gp -sNUMERO` oppure, durante l'esecuzione di Gp, utilizzare il comando `allocatemem` che raddoppia la quantità di memoria preesistente.

2) `primelimit` può essere incrementato eseguendo `gp` per mezzo del comando `gp -pNUMERO`.

2.2.3 Primi esempi

Vediamo ora qualche esempio di base.

1) i commenti sono identificati da una doppia barra:

```
gp > \\ questo e' un commento
```

2) l'assegnazione di un valore ad una variabile viene realizzata tramite il comando `=`

```
gp > x=2399393949
%1 = 2399393949
```

3) la stampa a video del valore di una variabile avviene usando il comando `print`:

```
gp > print(x)
2399393949
```

4) le operazioni di base hanno una sintassi analoga a quella usuale in matematica:

```
gp > x^3+77848848
%2 = 13813530083041663919441098197
gp > x*x
%3 = 5757091322497814601
gp > sqrt(x)
%4 = 48983.608983005733608881237291311990353
```

5) Abbiamo già usato la funzione (predefinita) `sqrt`. Altre funzioni predefinite che per noi saranno particolarmente importanti sono:

   (a) `factor(x)`: essa restituisce una matrice contenente la fattorizzazione completa del valore assunto dalla variabile `x` (nella prima colonna sono indicati i fattori primi p mentre nella seconda sono indicate le potenze α tali che $p^{\alpha} \,\|\, x$):

   ```
   gp > factor(x)
   %5 =
   [3 1]
   [193 1]
   [4144031 1]
   ```

   (b) `gcd(x,y)`: essa restituisce il valore del massimo comun divisore dei valori assunti dalle variabili `x` e `y`:

   ```
   gp > gcd(x,6390809)
   %11 = 193
   ```

6) in PARI/GP si ha accesso automatico ad una aritmetica intera a lunghezza arbitraria:

```
gp > x^100
%12 = 10236604580854796111054922719965911328088305080092010545849729312707202530054916325155314566497163742396846038235743464498265682423459321621679335231992877111693142793110061959289164984455088334070111401568120466629497211092294230465348653872418392493135399314695432024362566168086675228719473530277598914942769990271413462887220446688310035286604369736441315344873385600030821547372491384725211458557336245812157448383506112151580607425992894560905551739751241946859721981687258027593392585736124912622877120034263747960088955242309913265206309186598542440313477600485374856197993225066986199524654477882683937936607247635970824112034652460413043404148044422616789
```

```
   20459586122684069405133372939582455373982321530513563572148843
   67626243920561439985514861636806717000958612325827307939850643
   38236851663398817186304134315938459895136011538049041508756803
   84849357399599807356906609367935055356707303560838145918073083
   14498128154834616323761173000 1
```

2.2.4 HELP IN PARI/GP

La sorgente principale di informazioni sull'utilizzo di PARI/GP è la sua Guida per l'Utente (User's Guide). Una versione più stringata, ma spesso sufficiente per comprendere il funzionamento di una particolare funzione, si può anche ottenere con l'help in linea. Per visualizzare l'help in linea è sufficiente battere il comando `?` per ottenere la lista sotto riportata.

```
gp > ?
Help topics: for a list of relevant subtopics, type ?n for n in
   0: user-defined functions (aliases, installed and user functions)
   1: Standard monadic or dyadic OPERATORS
   2: CONVERSIONS and similar elementary functions
   3: TRANSCENDENTAL functions
   4: NUMBER THEORETICAL functions
   5: ELLIPTIC CURVES
   6: L-FUNCTIONS
   7: MODULAR SYMBOLS
   8: General NUMBER FIELDS
   9: Associative and central simple ALGEBRAS
  10: POLYNOMIALS and power series
  11: Vectors, matrices, LINEAR ALGEBRA and sets
  12: SUMS, products, integrals and similar functions
  13: GRAPHIC functions
  14: PROGRAMMING under GP
  15: The PARI community
Also:
  ? functionname (short on-line help)
  ?\  (keyboard shortcuts)
  ?.  (member functions)
Extended help (if available):
  ??  (opens the full user's manual in a dvi previewer)
  ??  tutorial / refcard / libpari (tutorial/reference
  card/libpari manual)
```

```
??  keyword    (long help text about "keyword"
from the user's manual)
??? keyword    (a propos: list of related functions).
```

Nel caso in cui si vogliano ottenere informazioni sulle funzioni della Teoria dei Numeri implementate in PARI/GP va eseguito il comando `?4`. In tal caso si otterrà la seguente risposta.

```
gp > ?4
addprimes   bestappr    bestapprPade   bezout
bigomega    binomial    charconj    chardiv
chareval    charker     charmul     charorder
chinese     content     contfrac    contfracpnqn
core  coredisc    dirdiv  direuler
dirmul  divisors    eulerphi    factor
factorback  factorcantor   factorff    factorial
factorint   factormod   ffgen   ffinit
fflog   ffnbirred   fforder     ffprimroot
fibonacci   gcd   gcdext  hilbert
isfundamental  ispolygonal  ispower     ispowerful
isprime     isprimepower   ispseudoprime  ispseudoprimepower
issquare    issquarefree   istotient   kronecker
lcm   logint  moebius     nextprime
numbpart    numdiv  omega   partitions
polrootsff  precprime   prime   primepi
primes  qfbclassno  qfbcompraw  qfbhclassno
qfbnucomp   qfbnupow    qfbpowraw   qfbprimeform
qfbred  qfbredsl2   qfbsolve    quadclassunit
quaddisc    quadgen     quadhilbert  quadpoly
quadray     quadregulator  quadunit    ramanujantau
randomprime  removeprimes   sigma   sqrtint
sqrtnint    stirling    sumdedekind  sumdigits
zncharinduce   zncharisodd  znchartokronecker  znconreychar
znconreyconductor  znconreyexp  znconreylog  zncoppersmith
znlog   znorder     znprimroot  znstar
```

Per accedere alle informazioni relative ad una specifica funzione va inserito il comando `?NOMEFUNZIONE`. Per esempio, nel caso desiderassimo leggere la descrizione della funzione che realizza un test di primalità, sarebbe sufficiente scrivere `?isprime` ed otterremmo seguente risposta:

```
gp > ?isprime
```

```
isprime(x,{flag=0}): true(1) if x is a (proven) prime number,
false(0) if not. If flag is 0 or omitted, use a combination of
algorithms. If flag is 1, the primality is certified by the
Pocklington-Lehmer Test. If flag is 2, the primality is
certified using the APRCL test.
```

Si noti che `isprime` può usare differenti algoritmi per rispondere al nostro quesito. La scelta tra gli algoritmi disponibili viene fatta mediante una variabile di comodo (in gergo chiamata `flag`). Questo tipo di soluzione è abitualmente usata in PARI/GP.

Alcuni testi di pseudoprimalità vengono invece eseguiti dalla seguente funzione:

```
gp > ?ispseudoprime
ispseudoprime(x,{flag}): true(1) if x is a strong pseudoprime,
false(0) if not. If flag is 0 or omitted, use BPSW test,
otherwise use strong Rabin-Miller test for flag randomly
chosen bases.
```

Usualmente PARI/GP indica con lo stesso nome funzioni diverse o che hanno argomenti di tipo diverso. Per esempio il massimo comun divisore di interi e di polinomi viene calcolato con la stessa funzione `gcd`.

Nel caso non si sia soddisfatti delle stringate indicazioni dell'help in linea, si può ottenere la pagina del manuale corrispondente ad una determinata funzione. Per esempio, per leggere la pagina del manuale corrispondente alla documentazione relativa alla funzione `isprime` è sufficiente scrivere `??isprime`.

```
gp > ??isprime
isprime(x,{flag = 0}):
true (1) if x is a (proven) prime number, false (0)  otherwise. ....
```

Nel caso in cui nel nostro sistema sia installato anche il software di composizione matematica TEX (con cui queste note sono composte), tale pagina verrà proposta direttamente in versione stampabile.

Per rimarcare nuovamente la differenza concettuale tra algoritmi di primalità e di fattorizzazione, riportiamo l'help in linea della funzione `factor` (di cui abbiamo precedentemente visto un esempio di uso):

```
gp > ?factor
factor(x,{lim}): factorization of x. lim is optional and  can be
set whenever x is of (possibly recursive)  rational type. If
lim is set return partial factorization, using primes up to lim.
```

La pagina del manuale di PARI/GP è molto più dettagliata e ne riportiamo solo un estratto:

```
gp > ??factor
factor(x,{lim}): General factorization function, where x is a
rational (including integers), a complex number with rational
real and imaginary parts, or a rational function (including
polynomials). The result is a two-column matrix: the first
contains the irreducibles dividing x (rational or Gaussian
primes, irreducible polynomials), and the second the
exponents. By convention, 0 is factored as 0^1....
```

2.3 Programmazione: alcuni comandi di base

Diamo ora alcuni esempi di programmazione in PARI/GP esaminando alcuni problemi di base. Useremo qui la funzione `lift` che ha il compito di consentire di passare da un elemento dello dell'insieme dei residui $\mathbb{Z}_n$ ad uno di $\mathbb{Z}$ (in generale gli elementi di questi due insiemi sono considerati di tipo diverso e, senza usare la funzione `lift`, non potrebbero essere confrontati).

```
gp > ?lift
lift(x,{v}): if v is omitted, lifts elements of Z/nZ to Z, of
Qp to Q, and of K[x]/(P) to K[x]. Otherwise lift only
polmods  with main variable v.
```

2.3.1 Leggere un file

Il comando `\r` consente di leggere il contenuto di un file.

Esempio 2.1. *Creiamo un file `pm.gp` che contiene la seguente linea (una funzione che ci consente di eseguire l'esponenziazione modulare presente nella forma debole del Piccolo Teorema di Fermat 1.7):*

```
  {fermat(a, n) = return (lift(Mod(a,n)^n))}
```

Per leggere questo file in PARI/GP è sufficiente usare `\r` :

```
gp > ?fermat
 ***  fermat: unknown identifier.
gp > \r pm
  \\ \r pm.gp fornisce lo stesso risultato
gp > ?fermat
```

```
fermat(a, n) = return (lift(Mod(a,n)^n))
gp > fermat(131,5)
%17 = 1
gp > fermat(131,7)
%18 = 5
gp > fermat(2,5)
%19 = 2
gp > fermat(2,4)
%20 = 0
gp > fermat(12,10)
%21 = 4
```

Nel caso il file `pm.gp` *venga modificato è sufficiente scrivere* `\r` *per ricaricarlo (nel caso si ometta il nome del file viene ricaricato l'ultimo file in ordine di tempo). Per esempio modificando*

```
return (lift(Mod(a,n)^n))
```

in `pm.gp` *con*

```
return (lift(Mod(a,n)^n)-a)
```

otteniamo una funzione che non ha particolare significato teorico-numerico, ma esemplifica l'uso di `\r`*:*

```
gp > \r
gp > fermat(2,4)
%22 = -2
```

2.3.2 Argomenti di funzioni e loro passaggio

Le funzioni di PARI/GP possono avere più di un argomento. Per esempio, la funzione

```
gp > {add(a, b, c, d)= return (a + b + c + d)}
gp > add(1,2,3,4)
%3 = 10
```

ha quattro argomenti. Se, nella chiamata della funzione, alcuni di essi vengono omessi, sono automaticamente posti uguali a 0.

```
gp > add(1,2)
%4 = 3
```

Se si vuole cambiare il valore di default di qualche variabile, è sufficiente includere tale informazione nella dichiarazione della funzione.

```
{add(a, b=-1, c=2, d=1)= return (a + b + c + d)}
```

In tal modo il valore di default per b è -1, per c è 2 e per d è 1. Tali valori verranno assegnati alle variabili b, c, d nel caso in cui nella chiamata della funzione `add` non sia specificato il valore dell'argomento corrispondente.

```
gp > add(1,2)
%6 = 6
gp > add(1)
%7 = 3
gp > add(1,2,3)
%8 = 7
```

2.3.3 Variabili locali

Nel linguaggio di scripting di PARI/GP tutte le variabili, a meno che non sia specificato il contrario, vengono considerate globali. Vediamo quindi come dichiarare variabili locali in PARI/GP.

Esempio 2.2. *La funzione* `errata` *somma gli interi* $1, 2, \dots, n$ *facendo un uso errato della variabile* `i`.

```
gp > {errata(n)=
  i=0;
  for(j=1,n, i=i+j);
  return(i)}
gp > errata(3)
%9 = 6
gp > i=4;
gp > errata(3);
gp > i
%10= 6   \\ ecco l'errore !!
```

Per usare correttamente le variabili si usa il comando `local`*: i è locale e dovrà essere dichiarata come tale.*

```
gp > {corretta(n)=
  local(i);
  i=0; for(j=1,n, i=i+j);
  return(i)}
```

```
gp > i=4;
gp > corretta(3)
%11 = 6
gp > i
%12 = 4
```

2.3.4 Come inserire i dati

Il comando `input` legge una espressione PARI/GP dalla tastiera o da un file. L'espressione viene valutata ed il risultato viene ritornato al programma.

Il tipo di dato della risposta dipende da come inseriamo il dato stesso. Per conoscere quale sia il tipo a cui appartiene un certo dato si utilizza il comando `type`. Ecco alcuni esempi:

```
gp > ?input
input(): read an expression from the input file or standard
input.
gp > s = input();
2+2
gp > s
%13 = 4   \\ attenzione: s non e' la stringa "2+2"
gp > s=input()
variabile
%14 = variabile
gp > type(s)
%15 = "t_POL" \\ PARI intende s come un polinomio
    \\ nella variabile ''variabile''
gp > s=input()
"ciao"
%16 = "ciao"
gp > type(s)
%17 = "t_STR"   \\ questa e' una stringa di caratteri
```

2.3.5 Scrivere in un file

Per salvare l'output di un programma PARI/GP in un file si usa il comando `write`:

```
gp >  ?write
write(filename,{str}*): appends the remaining arguments (same
output as print) to filename.
gp > write("provafile", "Qui Quo e Qua")
```

Il comando `write` appende la linea "Qui Quo e Qua" al fondo del file `provafile` (se il file non esiste, viene creato).

I dati prodotti in una sessione di lavoro sono anche disponibili nel file di log `pari.log` che si attiva con il comando `\l` e contiene la trascrizione integrale della sessione di lavoro.

```
gp > \l
  log = 1 (on)
gp > 2+2
%29 = 4
gp > \l
  log = 0 (off)
  [logfile was "pari.log"]
```

2.4 Esempi in PARI/GP

Elenchiamo alcuni altri esempi di calcoli teorico-numerici in PARI/GP.

2.4.1 Distribuzione dei primi

La prima domanda che ci poniamo è estremamente classica.
Domanda. *Quanti sono i numeri primi $\leq x$?*

Definita la funzione che "conta" quanti sono i numeri primi minori o uguali di un certo limite x nel modo seguente

$$\pi(x) = \mathrm{card}\{p \text{ primo tale che } p \leq x\},$$

sappiamo, grazie al Teorema dei Numeri Primi, che $\pi(x) \sim \mathrm{li}(x)$ per $x \to +\infty$.

Non possiamo certamente verificare un limite con il computer e quindi vogliamo solamente, per un fissato x passato in input, valutare quanti sono effettivamente i primi minori o uguali ad x. Sarà sufficiente contare le iterazioni di un ciclo `forprime`.

```
gp > pi(x, c=0) = forprime(p=2,x,c++); c;
```

Inoltre, per calcolare alcuni valori di $\pi(x)$ ed alcune sue approssimazioni, si può, per esempio, costruire un ciclo `for` come segue. Visto che vogliamo lavorare con primi un po' più grandi di quanto sono usualmente predefiniti, utilizziamo una nuova sessione di PARI/GP.

```
gp > pi(x, c=0) = forprime(p=2,x,c++); c;
 ? for(n=1,9,print("n= ",n*10^5," pi(n)= ", pi(n*10^5),
```

```
 " n/(log(n)-1)= ", n*10^5/(log(n*10^5)-1)))
n= 100000 pi(n)= 9592 n/(log(n)-1)= 9512.1001602462317324184
n= 200000 pi(n)= 17984 n/(log(n)-1)= 17847.46595229106177312
n= 300000 pi(n)= 25997 n/(log(n)-1)= 25836.37123394776422933
n= 400000 pi(n)= 33860 n/(log(n)-1)= 33615.64924810983906239
n= 500000 pi(n)= 41538 n/(log(n)-1)= 41246.08250334938142117
n= 600000 pi(n)= 49098 n/(log(n)-1)= 48761.91492985132947882
n= 700000 pi(n)= 56543 n/(log(n)-1)= 56185.02576693239107952
n= 800000 pi(n)= 63951 n/(log(n)-1)= 63530.54986227114849260
n= 900000 pi(n)= 71274 n/(log(n)-1)= 70809.54961226259999947
```

Possiamo usare PARI/GP anche per verificare, in qualche caso specifico, l'enunciato di alcuni famosi Teoremi.

- Uso del Teorema Cinese del Resto. La funzione `chinese` è predefinita in PARI/GP. La sua descrizione ed il suo uso sono i seguenti:

  ```
  gp > chinese(x,{y}): x,y being both intmods (or  polmods)
  computes z in the same residue classes as x and y.
  gp > chinese(Mod(2,3),Mod(3,5))
  %13 = Mod(8, 15)
  gp > chinese(Mod(8,15),Mod(2,7))
  %14 = Mod(23, 105)
  ```

  Se vogliamo risolvere un caso del problema dei generali cinesi (ogni plotone è formato al massimo da 1000 uomini; per capire quanti di essi sono presenti all'appello è sufficiente farli mettere in riga per 7, per 11 e poi per 13 e contare quanti ne rimangono nell'ultima riga; si usa poi il Teorema Cinese del Resto per determinare l'unica soluzione modulo $7 \cdot 11 \cdot 13 = 1001$ del sistema di tre congruenze ottenuto) dobbiamo "annidare" due volte la funzione `chinese` perché essa ammette solo due argomenti (chiaramente è possibile scrivere un programmino PARI/GP che utilizzi la funzione predefinita `chinese` e la estenda ad un sistema costituito da un numero arbitrario di congruenze lineari):

  ```
  gp > chinese(chinese(Mod(2,7),Mod(4,11)),Mod(11,13))
  %3 = Mod(37, 1001)
  ```

- Calcolo della funzione φ di Eulero.

  La funzione `eulerphi` è predefinita in PARI/GP. La sua descrizione ed il suo uso sono i seguenti:

```
gp > ?eulerphi
eulerphi(x): Euler's totient function of x.
gp >  eulerphi(2689*3^2*11^3)
%7 = 19514880
```

- Uso dell'Algoritmo di Euclide. Esso fornisce anche una relazione intera tra il massimo comun divisore di due interi e gli interi stessi. In letteratura tale relazione viene anche detta *Formula di Bézout*. Essa si ottiene mediante l'algoritmo Euclideo Esteso e, per tale ragione, la funzione PARI/GP che realizza tale calcolo viene chiamata `gcdext`.

```
gp > ?gcdext
gcdext(x,y): returns [u,v,d] such that d=gcd(x,y) and u*x+v*y=d.
gp > gcdext(1235,332)
%8 = [-25, 93, 1]
```

- Per realizzare l'esponenziazione modulare si può usare la funzione `Mod` combinata con l'operatore `^`. Vediamo alcuni esempi di esponenziazione modulare in PARI/GP.

```
gp > ?Mod
Mod(a,b): creates 'a modulo b'.
gp > Mod(11,2689)^2212
%10 = Mod(1493, 2689)
gp > Mod(1122348,3388827279)^223312
%12 = Mod(3029574879, 3388827279)
```

- La fattorizzazione di polinomi può essere calcolata mediante un'altra utile funzione di PARI/GP:

  `factormod(x,q).`

  Questa funzione consente di fattorizzare il polinomio `x` in $\mathbb{Z}_q$ e, per esempio, può essere usata per verificare l'irriducibilità su $\mathbb{F}_2$ del polinomio $x^8+x^4+x^3+x+1$. Questo polinomio è particolarmente importante perché è quello utilizzato nella descrizione di $\mathbb{F}_{256}$ nel metodo di cifratura simmetrico AES (Rijndael) (si veda, ad esempio il §5.6.6 di [6]).

```
gp > ??factormod
factormod(x,p,{flag=0}): factors the polynomial
x modulo the prime p, using Berlekamp. flag is optional,
```

```
and can be 0: default or 1: only
the degrees of the irreducible factors are given.

gp > factormod(x^4+x^2+1,2)
%5 = [Mod(1, 2)*x^2 + Mod(1, 2)*x + Mod(1, 2) 2]
  \\ x^4+x^2+1= (x^2+x+1)^2 su Z/2Z
gp > factormod(x^8 + x^4 + x^3 + x + 1,2)
%2 = [Mod(1, 2)*x^8 + Mod(1, 2)*x^4 +
  Mod(1, 2)*x^3 + Mod(1, 2)*x + Mod(1, 2) 1]
  \\ il polinomio x^8+x^4+x^3+x+1 e` irriducibile in F2[x]
```

CAPITOLO 3
INCONTRI

3.0 INTRODUZIONE INCONTRI

Riportiamo qui uno schema di cinque Incontri (di durata compresa tra le due ore e mezza e le tre ore ciascuno) relativi al progetto "Crittografia" scelto in Veneto dagli Istituti

- ITIS "Planck", Lancenigo (Tv);
- Liceo "Giorgione", Castelfranco;
- Liceo "Paleocapa", Rovigo;
- Liceo "Tron", Schio.

In particolare queste note riguardano lo schema relativo all'ITIS "Planck" di Lancenigo, Treviso.

L'intenzione è quella di fissare un problema o un concetto di base per ogni Incontro e di analizzarlo da un punto di vista teorico (per quanto può essere fatto a questo livello) presentando e discutendo con gli studenti alcuni risultati fondamentali. Lo scopo è quello di far giungere, mediante un lavoro autonomo ma guidato, gli studenti a comprendere un aspetto fondamentale (teorico o pratico) per ogni Incontro.

Di quando in quando faremo riferimento alle Dispensine riportate nella prima parte di queste note.

Per quanto riguarda i programmi, abbiamo inserito degli script scritti nel linguaggio di PARI/Gp. Sono facilmente leggibili essendo tale linguaggio di scripting essenzialmente un C semplificato e privato delle dichiarazioni di tipo. A differenza di un qualunque pseudocodice, essi sono eseguibili all'interno della shell di PARI/Gp. Quindi, oltre a consentire di fornire semplici esempi di programmazione validi dal punto di vista didattico, permettono anche di "testare" sul campo la loro consistenza.

Nel caso in cui qualche gruppo voglia utilizzare il C, ricordiamo che esiste un tool (GP2C) di PARI/Gp che consente la traduzione degli script in linguaggio C. A causa della grandezza degli interi coinvolti negli esempi significativi, se si vuole

utilizzare il C per sfruttare la maggiore velocità di un programma compilato rispetto ad uno script interpretato, si rende necessaria l'installazione di una libreria di programmi dedicati all'aritmetica intera a lunghezza arbitraria. Sebbene ciò sia assolutamente necessario se si desidera sviluppare un'applicazione utilizzabile nel mondo reale, dal nostro punto di vista ciò inficia la valenza didattica della presentazione perché appesantisce la presentazione degli algoritmi.

3.1 Primo Incontro

In questo primo incontro si dovrebbe per prima cosa fare un discorso introduttivo riguardante il ruolo della Crittografia nella vita di tutti i giorni facendo qualche semplice esempio (la cifratura delle comunicazioni dei telefoni cellulari GSM o la codifica del codice PIN del Bancomat).

Dopodiché si dovrebbero introdurre:

- il metodo di Cesare;
- il metodo di Vigenère.

In entrambi casi si possono utilizzare gli Esempi 1.5 e 1.6 delle Dispensine del Capitolo 1.

Per entrambi i metodi è previsto di fare lavorare gli studenti su qualche esempio facile per capire il tipo di ragionamenti richiesti dall'analisi delle frequenze.

A questo punto si presenta l'Esempio 1.7 che è trattato in dettaglio nelle Dispensine.

Può anche essere utile ricordare, se si lavora in ambiente LinuX, che alcuni comandi di tale sistema operativo (o di un qualunque altro "dialetto" UniX) sono utilizzabili per l'analisi delle frequenze. Ad esempio, dato il file Poe.txt il cui contenuto riproduce il crittogramma dell'Esempio 1.7 e che quindi è:

5 3 ‡ ‡ † 3 0 5)) 6 * ; 4 8 2 6) 4 ‡
.) 4 ‡) ; 8 0 6 * ; 4 8 † 8 ¶ 6 0))
8 5 ; 1 ‡ (; : ‡ * 8 † 8 3 (8 8) 5 *
† ; 4 6 (; 8 8 * 9 6 * ? ; 8) * ‡ (;
4 8 5) ; 5 * † 2 : * ‡ (; 4 9 5 6 * 2
(5 * – 4) 8 ¶ 8 * ; 4 0 6 9 2 8 5) ;
) 6 † 8) 4 ‡ ‡ ; 1 (‡ 9 ; 4 8 0 8 1 ;
8 : 8 ‡ 1 ; 4 8 † 8 5 ; 4) 4 8 5 † 5 2
8 8 0 6 * 8 1 (‡ 9 ; 4 8 ; (8 8 ; 4 (
‡ ? 3 4 ; 4 8) 4 ‡ ; 1 6 1 ; : 1 8 8 ;
‡ ? ;

possiamo utilizzare il comando

```
wc -w poe.txt
```

per calcolare il numero di simboli presente nel file (si faccia attenzione a utilizzare la codifica Unicode UTF-8 per il file di testo in questione). Per calcolare le occorrenze di ogni simbolo si può invece usare la sequenza di comandi

```
LC_ALL=C
export LC_ALL
cat poe.txt | tr ' ' '\n' | sort | uniq -c > frequenze.txt
```

che salva nel file frequenze.txt (se non esiste viene creato) quante volte ogni simbolo è presente nel file Poe.txt. È importante fornire le prime due direttive per ottenere una corretta identificazione dei caratteri secondo la codifica Unicode UTF-8.

Alla fine di tutto ciò dovrebbe essere abbastanza chiaro che da un punto di vista matematico la struttura importante è quella modulare e quindi si può fare qualche altro esempio facile (l'aritmetica dell'orologio analogico, per esempio).

A questo punto va data la definizione formale di Crittosistema.

Va anche fatto notare che gli esempi precedenti appartengono ai metodi *classici* in cui conoscere K_E è computazionalmente equivalente a conoscere K_D. Si ricordi che tali metodi sono detti anche *simmetrici* perché, per quanto sopra, il ruolo delle due chiavi è, dal punto della segretezza, interscambiabile (infatti in pratica vanno tenute entrambe segrete, perché rendere pubblica una implica consentire facilmente la determinazione dell'altra).

Alla fine dell'Incontro si nota che d'ora in poi cercheremo di fornire un po' della Matematica che serve per capire il funzionamento dei metodi *asimmetrici*: ossia quelli in cui non si può "facilmente" ricavare K_D dalla sola conoscenza di K_E. Si conclude notando che nei prossimi Incontri parleremo di numeri primi perché un metodo famoso e molto usato oggigiorno è basato sulle proprietà dei numeri primi.

3.1.1 Script PARI/Gp per Vigenère

Usando lo script PARI/Gp sotto riportato si possono costruire facilmente altri esempi per il codice di Vigenère con un alfabeto semplificato di 30 caratteri con solo lettere maiuscole, lo spazio bianco e qualche segno di interpunzione.

Prima di presentare lo script, osserviamo che il trattamento dei testi in PARI/Gp è un po' farraginoso. Per aggirare questo fatto ed utilizzare un semplice alfabeto di 30 caratteri (e non introdurre la codifica ASCII dei caratteri utilizzata da tutti i calcolatori) ci serviamo delle funzioni

```
gp> ??Vecsmall
Vecsmall({x = []}): ....  If x is a character string,  a vector
of individual characters in ASCII encoding is returned
(Strchr yields back the character string).

gp> ??Strchr
Strchr(x):  converts x to a string, translating each integer
into a character. Example:
```

```
gp> Strchr(97)
%1 = "a"
gp> Vecsmall("hello world")
%2 = Vecsmall([104, 101, 108, 108, 111, 32, 119, 111, 114, 108,
100])
gp> Strchr(%)
%3 = "hello world"
```

Per il controllo delle iterazioni usiamo il costrutto `for`.

```
gp > ??for
for(X = a,b,seq):
Evaluates seq, when the formal variable X goes from a to b.
Nothing is done if a > b. a and b must be in R.
```

Ecco lo script per il metodo di Vigenère. Si noti che gli equivalenti numerici delle lettere dell'alfabeto sono, in questo caso, gli interi appartenenti all'insieme $\{1,\ldots,30\}$.

```
/*********************************************************
*   Crittosistema di Vigenere
*   A. LANGUASCO e A. ZACCAGNINI per il PLS 2005-2007
*********************************************************/
{alfabeto=["A","B","C","D","E","F","G","H","I","J","K",
  "L","M","N","O","P","Q","R","S","T","U","V","W","X",
  "Y","Z",";",".","'"," "];
}
{da_lettera_a_numero(lettera)=local(j);
     for(j=1,30, if(alfabeto[j]==lettera,return(j)));
     error("input non valido.")
}

{VIGENERECIFRA(messaggio, chiave="", codifica="")
         = local(l,l1,i,j,k,C,w,M);
l=length(chiave);
l1=length(messaggio);
w=Vecsmall(chiave);
M=Vecsmall(messaggio);

/* Vecsmall trasforma una stringa in un vettore di numeri ASCII
corrispondenti. Per utilizzare il nostro alfabeto dovremo
```

```
tradurre poi ogni componente di questo vettore di nuovo in
una lettera singola (comando Strchr) e determinarne la
posizione nel nostro alfabeto*/

for(j=1,l1,
     if ((j % l) == 0, k=l, k= j % l);
     C= da_lettera_a_numero(Strchr(M[j])) +
          da_lettera_a_numero(Strchr(w[k]));
     if (C % 30 == 0, C = 30, C = C % 30);
     codifica=concat(codifica,alfabeto[C]);
);
print("Il messaggio codificato e`:");
print(codifica);
return(codifica);
}

{VIGENEREDECIFRA(codifica, chiave="", messaggio="")
         = local(l,l1,i,j,k,M,w,C);
l=length(chiave);
l1=length(codifica);
w=Vecsmall(chiave);
C=Vecsmall(codifica);
for(j=1,l1,
     if ((j % l) == 0, k=l, k= j % l);
     M=da_lettera_a_numero(Strchr(C[j])) -
          da_lettera_a_numero(Strchr(w[k]));
     if (M % 30 == 0, M = 30, M = M % 30);
     messaggio=concat(messaggio,alfabeto[M])
);
print("Il messaggio decodificato e`:");
print(messaggio);
return(messaggio);
}
```

Ed ecco un esempio del suo utilizzo:

```
gp > VIGENERECIFRA("INSICURO","FREQUENZA")
Il messaggio codificato e`:
OBXZXZBK
%7 = "OBXZXZBK"
gp > VIGENEREDECIFRA("OBXZXZBK","FREQUENZA")
```

```
Il messaggio decodificato e`:
INSICURO
%8 = "INSICURO"
```

3.2 Secondo Incontro

In questo secondo incontro si dovrebbe iniziare a parlare in dettaglio di $\mathbb{Z}_n$ facendo qualche semplice esempio ed aiutandosi con qualche figura esplicativa (per esempio la Figura 1.3).

Come secondo punto è necessario parlare dell'invertibilità degli elementi di $\mathbb{Z}_n$ facendo esempi con numeri piccoli ed osservando che da questo punto di vista la situazione "migliore" (cioè quella in cui si ha il numero massimo possibile di elementi invertibili) si ha quando n è primo.

A questo punto si fa notare che $\mathbb{Z}_n$, n composto, e $\mathbb{Z}_p$, p primo, sono oggetti un po' diversi tra loro. Ci si può aiutare con qualche tavola delle operazioni da costruire sul momento con numeri "piccoli". Più avanti esponiamo uno script per costruire una tabella per il prodotto in $\mathbb{Z}_n$.

Dovremmo poi spiegare come capire se un elemento è invertibile e come calcolarne l'inverso. Si introduce l'Algoritmo Euclideo e la Formula di Bézout e si fanno esempi della loro applicazione (si può utilizzare l'esempio delle Dispensine del Capitolo 1).

3.2.1 Costruzione della legge di gruppo di $(\mathbb{Z}_n, \cdot)$

Presentiamo un semplice script utile a generare esempi della legge di gruppo in $(\mathbb{Z}_n, \cdot)$.

```
/*******************************************************
*  TABELLA MOLTIPLICATIVA MODULO N
*  input: N ; identifica ZN
*  output: A: matrice dei risultati dei prodotti in ZN
*  A. LANGUASCO e A. ZACCAGNINI per il PLS 2005-2007
*******************************************************/
{Tabella(N) = local(A,i,j);
A=matrix(N,N);  /* definisce la tabella di dimensione NxN */
for(i=1,N,
       for(j=1,N,A[i,j]=lift(Mod(i*j-i-j+1,N)));
);
print("La tabella moltiplicativa modulo ",N," e`");
printtex(A);
print(A);
}

{Tantetabelle() = local(Y);
```

```
until(Y<>1,
       print ("Inserisci la dimensione: ");
             N=input();
             Tabella(N);
             Y=2;
             while ((Y<>0) && (Y <> 1),
             print ("Vuoi calcolare una nuova tabella?
                            (SI=1,NO=0) ");
             Y=input();
            );
      );
print ("Fine programma");
}
```

La funzione `Tabella` restituisce una tabella riguardante la legge di gruppo modulo N sia in formato TEX che con un output standard. Riportiamo qui solo quest'ultimo mentre la corrispondente descrizione in TEX (un po' modificata per esigenze estetiche) segue. Vediamo ora cosa accade se N è un composto (ad esempio $N = 12$).

```
gp > Tabella(12)
La tabella moltiplicativa modulo 12 e`
[0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 2 3 4 5 6 7 8 9 10 11]
[0 2 4 6 8 10 0 2 4 6 8 10]
[0 3 6 9 0 3 6 9 0 3 6 9]
[0 4 8 0 4 8 0 4 8 0 4 8]
[0 5 10 3 8 1 6 11 4 9 2 7]
[0 6 0 6 0 6 0 6 0 6 0 6]
[0 7 2 9 4 11 6 1 8 3 10 5]
[0 8 4 0 8 4 0 8 4 0 8 4]
[0 9 6 3 0 9 6 3 0 9 6 3]
[0 10 8 6 4 2 0 10 8 6 4 2]
[0 11 10 9 8 7 6 5 4 3 2 1]
```

A parte la prima riga identicamente nulla, esistono righe in cui sono presenti degli zeri in una posizione diversa dalla prima colonna. Ossia esistono dei divisori dello zero in $(\mathbb{Z}_{12}, \cdot)$: ad esempio, nella posizione identificata dalla terza riga e dalla settima colonna si ha uno zero perché questa posizione corrisponde a $2 * 6 = 12 \equiv 0 \mod 12$. Questi fatti sono più facilmente visibili usando la stampa in TEX :

$(\mathbb{Z}_{12},\cdot)$	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	10	11
2	0	2	4	6	8	10	0	2	4	6	8	10
3	0	3	6	9	0	3	6	9	0	3	6	9
4	0	4	8	0	4	8	0	4	8	0	4	8
5	0	5	10	3	8	1	6	11	4	9	2	7
6	0	6	0	6	0	6	0	6	0	6	0	6
7	0	7	2	9	4	11	6	1	8	3	10	5
8	0	8	4	0	8	4	0	8	4	0	8	4
9	0	9	6	3	0	9	6	3	0	9	6	3
10	0	10	8	6	4	2	0	10	8	6	4	2
11	0	11	10	9	8	7	6	5	4	3	2	1

Vediamo ora cosa accade se N è un primo (ad esempio $N = 11$). Come già sappiamo $\mathbb{Z}_{11}$ è un campo. Quindi ogni suo elemento non nullo ammette inverso moltiplicativo e non possono esistere in questo insieme divisori non banali dello zero. In effetti, a differenza del caso precedente, si può immediatamente notare che, nella tabella seguente, non esistono righe (a parte la prima) in cui sono presenti degli zeri in una posizione diversa dalla prima colonna. Per individuare l'inverso di un elemento fissato x, sarà sufficiente seguire la riga (o la colonna corrispondente) fino ad incontrare un 1. L'inverso di x sarà allora l'elemento sulla colonna (o riga) in cui compare 1. Ad esempio, l'inverso di 9 mod 11 è 5 perché un 1 è presente nella posizione $(10,6)$ corrispondente alla riga del 9 ed alla colonna del 5 (ed infatti $9\cdot 5 = 45 \equiv 1 \bmod 11$).

```
 gp > Tabella(11)
La tabella moltiplicativa modulo 11 e`
[0 0 0 0 0 0 0 0 0 0 0]
[0 1 2 3 4 5 6 7 8 9 10]
[0 2 4 6 8 10 1 3 5 7 9]
[0 3 6 9 1 4 7 10 2 5 8]
[0 4 8 1 5 9 2 6 10 3 7]
[0 5 10 4 9 3 8 2 7 1 6]
[0 6 1 7 2 8 3 9 4 10 5]
[0 7 3 10 6 2 9 5 1 8 4]
[0 8 5 2 10 7 4 1 9 6 3]
[0 9 7 5 3 1 10 8 6 4 2]
[0 10 9 8 7 6 5 4 3 2 1]
```

Anche in questo caso, questi fatti risultano più facilmente visibili usando la stampa in TEX del risultato:

$(\mathbb{Z}_{11},\cdot)$	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	10
2	0	2	4	6	8	10	1	3	5	7	9
3	0	3	6	9	1	4	7	10	2	5	8
4	0	4	8	1	5	9	2	6	10	3	7
5	0	5	10	4	9	3	8	2	7	1	6
6	0	6	1	7	2	8	3	9	4	10	5
7	0	7	3	10	6	2	9	5	1	8	4
8	0	8	5	2	10	7	4	1	9	6	3
9	0	9	7	5	3	1	10	8	6	4	2
10	0	10	9	8	7	6	5	4	3	2	1

3.2.2 Algoritmo Euclideo Esteso

Descriviamo ora l'Algoritmo Euclideo con Formula di Bézout (anche detto Algoritmo Euclideo Esteso) in pseudocodice e forniamo uno script PARI/Gp. Si noti che in PARI/Gp esiste già una funzione predefinita `gcd` che fornisce il massimo comun divisore di due interi:

```
gp> ?gcd
gcd(x,{y}): greatest common divisor of x and y.
```

Esiste anche la funzione che esegue l'Algoritmo Euclideo Esteso:

```
?gcdext
gcdext(x,y): returns [u,v,d] such that d=gcd(x,y) and u*x+v*y=d.
```

Supponiamo che m,n, con $m \geq n$, siano due naturali positivi. Lo schema dell'Algoritmo di Euclide è:

(1) Si pone $r_{-1} \leftarrow m$, $r_0 \leftarrow n$, $k \leftarrow 0$;

(2) se $r_k = 0$ allora $r_{k-1} = (m,n)$; l'algoritmo termina;

(3) si divide r_{k-1} per r_k trovando due interi q_{k+1} ed r_{k+1} con

$$r_{k-1} = q_{k+1} r_k + r_{k+1},$$
$$0 \leq r_{k+1} < r_k.$$

(4) Si pone $k \leftarrow k+1$.

(5) Si torna al passo (2).

L'algoritmo termina poiché la successione $(r_k) \subseteq \mathbb{N}$ è monotona decrescente. Per determinare λ e μ costruiamo due successioni a_k e b_k: i valori iniziali sono

$$a_{-1} = 1, \quad b_{-1} = 0, \quad a_0 = 0, \quad b_0 = 1.$$

Poi si calcolano a_k e b_k mediante

$$a_k = a_{k-2} - q_k a_{k-1}, \qquad b_k = b_{k-2} - q_k b_{k-1}. \tag{3.1}$$

Queste due successioni hanno la proprietà che $r_k = a_k m + b_k n$ per ogni $k > 0$ ed in particolare, se $r_{K+1} = 0$, per $k = K$ e quindi

$$r_K = (m,n) = a_K m + b_K n.$$

Il numero di moltiplicazioni o divisioni necessarie per l'esecuzione è $O(\log m)$. Questo segue dalle osservazioni immediatamente successive alla Proposizione 1.1.

Si può giungere allo stesso risultato osservando che il caso peggiore è quello in cui $q_k = 1$ perché in tal modo i resti r_k sono i più grandi possibile. In tal caso la successione f_k definita per ricorrenza da $f_0 = 0$, $f_1 = 1$, ed $f_{k+2} = f_{k+1} + f_k$ (numeri di Fibonacci) costituisce l'insieme delle coppie (f_{k+2}, f_{k+1}) che, fra *tutte* le coppie di interi entrambi $\le f_{k+2}$, richiede più passi dell'Algoritmo Euclideo Esteso, prima di raggiungere il resto 0, e ne richiede esattamente k. In sostanza, dalla definizione si deduce che $f_{k+2} = 1 \cdot f_{k+1} + f_k$, e cioè il quoziente fra due numeri di Fibonacci consecutivi vale sempre 1, se $k \ge 2$, mentre il resto è precisamente il numero di Fibonacci immediatamente precedente. Dunque abbiamo la catena di uguaglianze $(f_{k+2}, f_{k+1}) = (f_{k+1}, f_k) = (f_k, f_{k-1}) = \cdots = (f_2, f_1)$, ed a questo punto il resto successivo vale 0 poiché $f_1 = 1$. Ricordando la Formula di Binet (che si può dimostrare per induzione)

$$f_k = \frac{1}{\sqrt{5}} \Big\{ \Big(\frac{1+\sqrt{5}}{2} \Big)^k - \Big(\frac{1-\sqrt{5}}{2} \Big)^k \Big\},$$

possiamo concludere, dato che $|(1-\sqrt{5})/2| < 1$, che, per k grande, f_k è molto vicino a $5^{-1/2}\big((1+\sqrt{5})/2\big)^k$.

Dato che f_k cresce con velocità essenzialmente esponenziale di base $a > 1$, questo giustifica la nostra affermazione che il numero di iterazioni necessarie all'Algoritmo di Euclide non supera, in ordine di grandezza, il logaritmo del più grande dei due; possiamo pertanto concludere che il numero di iterazioni dell'Algoritmo Euclideo Esteso è al più $O(\log f_{k+2})$ che è $O(k)$; altri dettagli si trovano sul testo [6].

3.2.3 Script PARI/Gp per l'Algoritmo Euclideo

Per il controllo delle iterazioni useremo il costrutto `while`.

```
gp > ??while
while(a,seq): while a is nonzero evaluate the expression
sequence seq. The test is made before evaluating the
seq, hence in particular if a is initially equal to zero the
seq will not be  evaluated at all
```

Ecco uno script in PARI/Gp per l'Algoritmo Euclideo Esteso.

```
/*********************************************************
*   Algoritmo Euclideo Esteso.
*   A. LANGUASCO e A. ZACCAGNINI per il PLS 2005-2007
*********************************************************/
/* input: n,m - numeri di cui si ricerca un il massimo
 * comun divisore, m>n. Se m<n i due numeri vengono scambiati.
 * output: d=(m,n) e due interi lambda, mi tali che
 * d=(lambda)m+(mi)n     ***/

{EuclideEsteso(m,n) = local (M,N,q,r,a,b,a0,b0,A,B,
                             lambda,mi,app,iterazioni);
/*  n e m devono essere positivi */
if(((n <= 0) || (m<=0)), error("I due numeri in input
devono essere entrambi positivi !!"));

/* se n=m il risultato e' banale  */
if(n == m, d=n; a=1;b=0;
          print("I due numeri sono uguali");
          print("d=",d, " a=",a, " b=",b);
          return     );

/* se m<n scambio i due numeri  */
if(m < n, app=m; m=n; n=app);

/* Inizializzazione delle successioni a_k e b_k
A=a_{-1} = 1, B= b_{-1} = 0,  a_0 = 0,  b_0 = 1.  ***/
A=1; a0=0;
B=0; b0=1;
```

```
/* Inizializzazione delle altre variabili  */
iterazioni=0;
M=m;  /* serve solo a ricordare il valore di m */
N=n;  /* serve solo a ricordare il valore di n */

while (n!=0,
 r= m %n ;   /* resto della divisione intera di m per n */
 q= m \ n;  /* quoziente della divisione intera di m per n */
 a=A-q*a0;      /* a_k = a_{k-2} - q_k a_{k-1} */
 b=B-q*b0;      /* b_k = b_{k-2} - q_k b_{k-1} */
 m=n;           /* scambio il ruolo di m e n, e di n e r per */
 n=r;    /* fare al passo successivo la divisione di n per r */
 A=a0;/* per il passo successivo serve a_{k-1} va in a_{k-2} */
 a0=a;/* per il passo successivo serve a_{k} va in a_{k-1} */
 B=b0;/* per il passo successivo serve b_{k-1} va in b_{k-2} */
 b0=b;/* per il passo successivo serve b_{k-1} va in b_{k-2} */
 iterazioni=iterazioni+1      /*conto il numero di iterazioni*/
);

/* alla fine del while si ha r_(K+1)=0; allora d e` il resto
 * del passo precedente che e` immaganizzato in m; i
 * moltiplicatori sono anch'essi quelli del passo precedente
 * e sono rispettivamente immagazzinati in  A e B   ****/
d=m;
lambda=A;
mi=B;

/* Output dei risultati */
print("Il massimo comun divisore di m=",M, " e n=", N, " e`");
  print("d=",d);
  print("I moltiplicatori sono rispettivamente lambda=",
     lambda, " e mi=",mi);
  print("Il numero totale di iterazioni e` ", iterazioni);
  return;}
```

Ed ecco un esempio del suo utilizzo:

```
gp> EuclideEsteso(1235,332)
Il massimo comun divisore di m=1235 e n=332 e'
d=1
I moltiplicatori sono rispettivamente lambda=-25 e mi=93
```

```
Il numero totale di iterazioni e' 7
```

3.3 Terzo Incontro

In questo terzo incontro si dovrebbe cominciare a parlare di fattorizzazione e primalità di interi. Visto che non è possibile sviluppare la teoria della Complessità Computazionale se non per cenni, si può cominciare l'Incontro utilizzando la funzione `factor`

```
gp > ?factor
factor(x,{lim}): factorization of x. lim is optional and  can be
set whenever x is of (possibly recursive)  rational type. If
lim is set return partial factorization, using primes up to lim.
```

e la funzione `isprime` di PARI/Gp

```
gp > ?isprime
isprime(x,{flag=0}): true(1) if x is a (proven) prime number,
false(0) if not. If flag is 0 or omitted, use a combination of
algorithms. If flag is 1, the primality is certified by the
Pocklington-Lehmer Test. If flag is 2, the primality is
certified using the APRCL test.
```

Il suggerimento è di applicarle entrambe al numero RSA-220. Esso è un numero di 220 cifre decimali che è stato fattorizzato nel 2016 e che è stato una delle "sfide" proposte dagli RSALabs `http://www.rsasecurity.com/rsalabs/`. Tale numero è stato costruito per "resistere" alle strategia di fattorizzazione note e la sua fattorizzazione costituisce ancora un problema che richiede un grande quantità di tempo (in giorni) per i computer "domestici". Il numero in questione è

```
RSA220 = 2260138526203405784941654048610197513508038915
719776718321197768109445641817966676608593121306582577 2
506315628866769704480700018111497118630021124879281994 8
748206607013106658664608332798280356037920539198013994 6
496955261
```

e sappiamo che ha i seguenti fattori primi

```
p=6863656412267566274382371499288437800130842239979164
844621244993321541061441464266793821364420842019205499968 7
q=3292907439486349812049301549212935291916455196536233 9
524626860511692903493094652463337824866390738191765712603
```

Per prima cosa si usa subito `factor` su RSA220 e si vede che PARI/Gp comincia a "lavorare" ma non fornisce una risposta immediata. Dopodiché si fa partire una nuova sessione di PARI/Gp e si usa `isprime` su RSA220. Si osserverà che questa seconda sessione, malgrado il computer sia nel frattempo impegnato anche nella prima, risponde in modo praticamente immediato che RSA220 è composto.

Alla fine dell'Incontro si potrà constatare come la prima sessione sia ancora in esecuzione ...

Dopo la risposta di `isprime` dovrebbe essere già abbastanza chiaro, perlomeno in questo esempio, che, sebbene il problema della primalità e quello della fattorizzazione possano ingenuamente sembrare lo stesso, in realtà da un punto di vista computazionale sono profondamente diversi (alla fine dell'Incontro ciò sarà ancora più chiaro, visto che la prima sessione non sarà ancora terminata).

A questo punto va sfruttato il "clima" creato dall'esempio pratico precedente per far fare agli studenti un "atto di fede": ossia la primalità si può attualmente risolvere con "pochi" calcoli (è un problema computazionalmente "facile") mentre per la fattorizzazione (a parte alcuni casi particolari) non sono noti algoritmi risolutivi che necessitano di "pochi" calcoli (è un problema computazionalmente "difficile").

Ora bisognerebbe parlare del Piccolo Teorema di Fermat 1.7. A seconda dei gusti si può utilizzare la dimostrazione combinatoria oppure quella più formale. In entrambi i casi è necessario richiamare sia la forma forte (se $(a,p) = 1$ allora $a^{p-1} \equiv 1 \bmod p$) che quella debole ($a^p \equiv a \bmod p$ per ogni $a \in \mathbb{Z}$).

Poi va dato (senza soffermarsi su dimostrazioni) l'enunciato del Teorema di Eulero-Fermat e dei risultati necessari per dimostrare la correttezza della decifratura del metodo RSA. Ciò è necessario per far capire cosa accade se si lavora modulo un intero composto n (come in RSA).

Tali risultati sono esposti nelle Dispensine del Capitolo 1 e sono il Teorema 1.8 e la Proposizione 1.8.

Per poter parlare del Teorema di Eulero-Fermat 1.8 e degli altri risultati menzionati precedentemente, bisognerà prima chiedere agli studenti di avere un poco di pazienza per definire la funzione φ di Eulero; per le applicazioni successive è sufficiente soffermarsi sul caso $n = pq$. Il ragionamento che serve è esposto in generalità nella Proposizione 1.6 e può essere proposto agli studenti, nel caso specifico $n = pq$, come esercizio da fare in modo discusso con i docenti.

3.4 Quarto incontro

In questo quarto incontro si dovrebbe ritornare sui numeri primi spiegando come fare a "produrli" e come invece è difficile determinarli all'interno della fattorizzazione di un intero dato.

A tal scopo, anziché parlare di pseudoprimalità (che richiederebbe, pensando al metodo di Miller-Rabin, perlomeno il concetto di radice quadrata modulare), pensiamo che sia preferibile introdurre due semplici algoritmi:

1. il Crivello di Eratostene per la costruzione di tabelle di primi (si veda il Capitolo 1);

2. il metodo della divisione per tentativi (ricerca di un fattore di un intero).

Oltre a presentarli va specificato anche quanto essi "costino" in termini di "onerosità di calcolo". Il primo determina *tutti i primi* in $[1,n]$ con una complessità $O(n\log\log n)$ divisioni. Infatti per ogni $p|n$ vengono essenzialmente eseguite $\left\lfloor \frac{n}{p} \right\rfloor$ divisioni; quindi va calcolato

$$\sum_{p\le n^{1/2}} \left\lfloor \frac{n}{p} \right\rfloor \le \sum_{p\le n^{1/2}} \frac{n}{p} = n \sum_{p\le n^{1/2}} \frac{1}{p} \approx n\log\log n,$$

per uno dei Teoremi di Mertens. Euristicamente, quindi, ogni primo $p \le n$ determinato in questo modo ha un costo computazionale "medio" di $\approx \log\log n$ divisioni.

Il secondo determina il *più piccolo m fattore di un intero* con complessità $O(m)$ divisioni. Si osservi che, nel caso peggiore, essa diviene $O(\sqrt{n})$ divisioni.

Ricordiamo che la divisione di due interi $\le n$ può essere eseguita mediante $O\big(\log^2 n\big)$ operazioni elementari (ossia operazioni di somma o prodotto, con riporto, su un singolo bit; esse sono considerate praticamente istantanee e vengono utilizzare come unità di misura del calcolo delle complessità computazionali degli algoritmi che lavorano sugli interi).

Descriviamo qui il metodo della divisione per tentativi essendo il Crivello di Eratostene descritto nel Capitolo 1.

3.4.1 Divisione per tentativi

Si può dimostrare che un numero intero $n \ge 2$ è primo verificando direttamente la definizione, cioè verificando che nessuna delle divisioni di n per gli interi $2 \le m \le n-1$ è esatta. Poiché se $n = mr$ uno fra m ed r è necessariamente $\le \sqrt{n}$, è sufficiente effettuare $O(\sqrt{n})$ divisioni. Avendo una lista dei numeri primi $\le \sqrt{n}$ è sufficiente provare a dividere n per ciascuno di questi numeri primi, ma in ogni caso il numero

delle divisioni necessarie non è significativamente più piccolo di $\sqrt{n}$. L'algoritmo ha una complessità computazionale di $O(\sqrt{n})$ divisioni.

Come esempio numerico può essere usato $H = 8\,618\,843\,833$.

Esempio 3.1. *Se si considera il numero* $H = 8\,618\,843\,833$*, si devono fare circa* 44844 *divisioni per ottenerne la scomposizione in fattori. Naturalmente è possibile "risparmiare" molte di queste divisioni osservando che è inutile tentare di dividere* H *per un intero pari, ma in ogni caso il numero di divisioni da fare, anche avendo a disposizione la lista di tutti i numeri primi fino a* $\lfloor\sqrt{H}\rfloor$*, è circa* 10000*.*

Abbiamo già parlato della differenza "pratica" del tempo di calcolo di `isprime` e di `factor` nell'Incontro 3.3.

3.4.2 Script PARI/Gp per il Crivello di Eratostene ed il Trial Division

Infine alleghiamo qui tre script in PARI/Gp.

1. Divisione per tentativi; nella versione in cui si divide per solo per gli interi dispari minori o uguali a $\sqrt{n}$.

```
/*********************************************************
*   Divisione per Tentativi
*   A. LANGUASCO e A. ZACCAGNINI per il PLS 2005-2007
*********************************************************/

/* input: n - numero di cui si ricerca un fattore
 * output: FALSO o VERO (VERO significa che n e' primo)
 * Dopo aver verificato che n non e' pari si prova a
 * dividere per ogni intero dispari minore o uguale
 * della radice quadrata di n. ******/

{Trialdivision(n) = local (TRUE,FALSE, iterazioni);
TRUE=1;
FALSE=0;
iterazioni=0;
 if(n % 2 == 0,
 print("Il numero ", n," e' pari");
 iterazioni=iterazioni+1;
 print("Il numero totale di iterazioni e' ", iterazioni);
 return(FALSE);
```

```
 );
forstep (m=3,floor(sqrt(n)),2,
 iterazioni=iterazioni+1;
 if (n%m==0,
 print(m," divide ",n);
 print("Il numero totale di iterazioni e` ", iterazioni);
 return(FALSE)
 )
);
print(n, " e` un numero primo");
print("Il numero totale di iterazioni e` ", iterazioni);
return(TRUE);
}
```

Lo script Trialdivision applicato al numero H fornisce i seguenti risultati

```
gp >Trialdivision(8618843833)
89689 divide 8618843833
Il numero totale di iterazioni e` 44844
%44 = 0
gp > ##
***   last result computed in 21 ms.
```

2. Crivello di Eratostene in una versione semplice. Ossia senza inserire vettori di variabili booleane o sfruttare proprietà delle congruenze.

```
/*********************************************************
*  CRIVELLO DI ERATOSTENE
*  input: L - limite entro il quale ricercare i primi.
*  output: vettore contenente tutti i primi tra 2 e L.
*       A. LANGUASCO e A. ZACCAGNINI per il PLS 2005-2007
*********************************************************/
{Erat(L) = local(a,B,B1,i,j,k ,l,primi,primiout);

/* inizializziamo un vettore di L-1 componenti con gli
* interi tra 2 e L e definiamo un altro vettore della
* stessa dimensione in cui memorizzeremo i primi  */

 B=L-1;
```

```
 a=vector(B);
 primi=vector(B);
 for(i=1, B, a[i] = i+1);

/* Intendiamo come marcato un intero la cui corrispon-
 * dente componente di a e' zero. La componente viene
 * marcata a zero se l'intero contenuto e' un multiplo di
 * uno degli interi precedenti */

 i = 1;
 l=1;
 B1=floor(sqrt(L));
 while(i <= B1,

/* "saltiamo" gli interi gia' marcati */
   while( a[i] == 0, i = i + 1);

/* alla fine del while, l'intero contenuto nella
 * posizione a[i]  deve essere un primo perche'
 * non e' diviso da nessun intero piu' piccolo.
 * Allora lo memorizziamo nella prima posizione
 * disponibile del vettore primi     */

   primi[l]=a[i];
   l=l+1;

/* marchiamo adesso i multipli dell'intero contenuto in
* a[i]=i+1, ossia azzeriamo le posizioni corrispondenti
*agli interi j(i+1), per j che varia fino a L/i+1. Queste
* posizioni hanno indice j(i+1)-1       */

   for(j=2, floor(L/(i+1)), a[(i+1)*j-1] = 0);

/* passiamo ora a studiare la primalita' dell'intero
 * successivo incrementando la variabile che governa
 * il while piu' esterno */

  i=i+1;
  );
```

```
 /* inseriamo nel vettore dei primi i primi piu' grandi
  * di radice di L */

 for (j=i+1,B, if ((a[j]<>0), primi[l]=a[j];l=l+1));

 /* Sappiamo anche  quanti sono i primi minori o uguali
 * a L (sono l-1) e prepariamo l'output del vettore dei
 * primi */

 print("Il numero di primi minori o uguali a ", L,
           " e': ", l-1);

 /* restituiamo in output il vettore dei primi */
 primiout=vector(l-1);
 for(j=1,l-1,primiout[j]=primi[j]);
 print("I numeri primi minori o uguali a ", L, " sono dati
      dal vettore: ");
 return(primiout);
}
```

Il crivello di Eratostene ha un'occupazione di memoria notevole e quindi PARI/Gp non riesce a trattare il numero H (a meno di allocare una quantità ancora superiore di memoria). Il limite di utilizzo senza incrementare la memoria è circa di 6 cifre decimali.

```
gp > Erat(200111)
Il numero di primi minori o uguali a 200111 e': 17993
I numeri primi minori o uguali a 200111 sono:
%33 = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107,
109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167,
173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229,
233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283,
293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359,
367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431,
433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491,
499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571,
577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641,
643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709,
.................]
```

```
? ##
 ***   last result computed in 352 ms.
```

3. Una versione leggermente più performante del Crivello di Eratostene in cui si è usato un vettore di bit invece che un vettore di interi. La seguente versione è più veloce di circa il 37% rispetto alla precedente.

```
/*******************************************************
*  CRIVELLO DI ERATOSTENE
*  input: L - limite entro il quale ricercare i primi.
*  output: vettore contenente tutti i primi tra 2 e L.
*       A. LANGUASCO e A. ZACCAGNINI per il PLS 2005-2007
*******************************************************/

{Eratsmart(L) = local(a, l, i, l1, u, j, npi, k, primi,
                   primiout);

/* inizializziamo un vettore di L componenti booleane. In
* Gp non e' possibile farlo direttamente. Allora poniamo a
* uguale al  numero che ha L cifre binarie uguali a 1 */

 a=binary(2^L-1);
 l=length(a);

 /* intendiamo come marcato un intero j per cui la
  * relativa componente di a e' zero. La componente
  * a[j] viene posta uguale a zero se l'intero j e' un
  * multiplo di uno degli interi precedenti. */

 i = 2;
 l1= floor(sqrt(l));
 while(i <= l1,

 /* "saltiamo" gli interi gia' marcati  */

  while(a[i] == 0,i = i + 1);

  /* alla fine del while, a[i]=1 indica che i deve essere
   * un primo perche' non e' diviso da nessun intero
   * piu' piccolo. Marchiamo adesso i multipli dell'intero
```

```
   * i ossia azzeriamo le componenti di a corrispondenti
   * agli interi j*i, per j che varia fino a [l/i]. ***/

   u= l \ i;
   for(j=i, u , a[i*j] = 0);

    /* passiamo ora a studiare la primalita' dell'intero
    * successivo incrementando la variabile che governa
    * il while piu' esterno */

 i=i+1;
 );

 /* restituiamo in output il vettore i primi   */

 primi=vector(l);
 npi=1;
 for (k=2,l, if ((a[k]<>0), primi[npi]=k;npi=npi+1));

 /* Sappiamo anche  quanti sono i primi minori o uguali
  * a L (sono npi-1) */

 print("Il numero di primi minori o uguali a ", L,
             " e': ",  npi-1);
 primiout=vector(npi-1);
 for(k=1, npi-1, primiout[k]=primi[k]);
 print("I numeri primi minori o uguali a ", L, " sono dati
     dal vettore: ");
 return(primiout);
}
```

Ed ecco il calcolo sullo stesso numero precedente.

```
gp > Eratsmart(200111)
Il numero di primi minori o uguali a 200111 e': 17993
I numeri primi minori o uguali a 200111 sono dati dal
vettore:
%33 = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107,
109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167,
```

```
173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229,
233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283,
293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359,
367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431,
433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491,
499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571,
577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641,
643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709,
.................]
? ##
  ***   last result computed in 223 ms.
```

3.5 Quinto incontro

In questo quinto incontro si dovrebbe cominciare richiamando la definizione di Crittosistema e facendo vedere che è possibile comunicare senza preventivamente scambiarsi le chiavi (come negli esempi del primo Incontro) se si suppone che esiste "un lucchetto inviolabile": ossia si presenta il metodo del doppio lucchetto.

In tal modo si fa capire che si può comunicare con una persona pur mantenendo con essa un certo livello di segretezza nei parametri della comunicazione.

A questo punto si richiama la definizione di Crittografia *asimmetrica* e poi si procede a definire "passo passo" utilizzando contemporaneamente la lavagna (per le definizioni) e PARI/Gp (per far vedere che i parametri possono essere calcolati con tempi "praticamente" abbordabili) i parametri fondamentali del metodo RSA. In questo punto si possono far calcolare agli studenti tutti i parametri necessari replicando il lavoro che un docente fa con PARI/Gp.

Un possibile schema di lavoro è qui riportato:

3.5.1 RSA con PARI/Gp

Useremo l'operatore di negazione `!` applicato a `=` (che restituisce 0 se due quantità sono diverse e 1 se sono uguali) e le seguenti due funzioni.

```
gp >?nextprime
? nextprime
nextprime(x): smallest pseudoprime >= x.

gp > ?random
random({N=2^31}): random object, depending on the type of N...
```

Inoltre è necessario fare le potenze modulari; in PARI/Gp esiste già questa funzione che si realizza tramite le funzioni `Mod` e `lift`.

```
gp> ?Mod
Mod(a,b): creates 'a modulo b'.
gp> ?lift
lift(x,{v}): if v is omitted, lifts elements of Z/nZ to Z, of Qp to Q,
and of K[x]/(P) to K[x]. Otherwise lift only polmods with main
variable v.
```

- Generazione di p e q.

```
p=nextprime(random(10^50))
```

```
%2 = 22005109792022134749492304074618983912551252046861
gp > isprime(%2)
%3 = 1
gp > q=nextprime(random(10^60))
%5 = 46115964053880060094485305924606274945238607476053
53656267853
gp > isprime(%5)
%6 = 1
```

- Calcolo di n.

```
gp > n=p*q
%7 = 10147868521705768912906357852166732520259672369610
974 69715018803977036992478064186314786954326020738832
3859433
```

- Calcolo di $\varphi(n)$.

```
gp > phin=(p-1)*(q-1)
%8 = 10147868521705768912906357852166732520259672369610
513537509627234059633049586646308094342538201534283415
544720
```

- Calcolo di e.

```
gp >  e=random(n)
%9 = 732201225334401894703078938632653293576002219538603
748236461978823051711778253268198635241742987529750559
6696
gp > while(gcd(e,phin)!=1,e=e+1)
gp > e
%10 = 732201225334401894703078938632653293576002219538603
748236461978823051711778253268198635241742987529750559
6697
```

- Calcolo di $d \equiv e^{-1} \bmod \varphi(n)$.

```
gp > d = lift(Mod(e,phin)^(-1));
gp > (e*d)%phin
%9 = 1
```

```
gp > d
%13 = 767641219086839958440527925874666900211053767589 3
9173566334605996179541737969552257520404748664265575 85
37193
```

- Calcolo della lunghezza massima del blocco di testo codificabile.

```
gp > log(n)/log(30)
%11 = 73.796497398886431714670067961393849992
```

Usando un alfabeto di 30 caratteri possiamo quindi codificare in un blocco unico un testo avente al più 73 caratteri. Codifichiamo "PADOVA":

- Calcolo dell'equivalente numerico in base 30 di PADOVA usando l'alfabeto: "A","B","C","D","E","F","G","H","I","J","K","L","M","N","O","P", "Q","R","S","T","U","V","W","X","Y","Z",",",".","' ",". ".

  Si noti che, a differenza dell'alfabeto usato nel metodo di Vigenère, gli equivalenti numerici delle lettere dell'alfabeto sono, in questo caso, gli interi appartenenti all'insieme $\{0,\dots,29\}$.

```
gp > m=15*30^5+0*30^4+3*30^3+14*30^2+21*30^1+0*30^0
%12 = 364594230
```

- Definizione della funzione di cifratura.

```
gp > E(x)=lift(Mod(x,n)^e)
```

- Definizione della funzione di decifratura.

```
gp > D(x)=lift(Mod(x,n)^d)
```

- Codifica di PADOVA.

```
gp > messaggio_segreto = E(m)
%18 = 84391967286491764060498053664729875964514329970448
78475692460047409901642140614293270114149067170870027
254030
```

- Decodifica.

```
gp > D(messaggio_segreto)
%14 = 364594230
```

La procedura è stata sviluppata correttamente poiché l'equivalente numerico della decodifica è uguale a quello di PADOVA.

3.5.2 Script PARI/Gp per RSA

A questo punto si può "lasciare giocare" gli studenti con PARI/Gp oppure si può tentare di guidarli per realizzare uno script che "automatizzi" alcune delle operazioni necessarie.

Esistono chiaramente molti modi per raggiungere questo scopo. Qui ne proponiamo uno che potete seguire o meno a seconda dei vostri gusti.

Il problema principale è che non sappiamo a priori quanto è lungo il messaggio. Esso può essere risolto in un modo veloce, ma ingenuo, semplicemente concatenando al fondo del messaggio (nel caso in cui esso sia più corto del massimo blocco di testo codificabile) una serie di spazi bianchi.

Vediamo adesso un possibile script. Facciamo ancora notare che, a differenza dell'alfabeto usato nel metodo di Vigenère, gli equivalenti numerici delle lettere dell'alfabeto sono, in questo caso, gli interi appartenenti all'insieme $\{0,\dots,29\}$.

```
/*********************************************************
*   RSA
*   A. LANGUASCO e A. ZACCAGNINI  per il PLS 2005-2007
*********************************************************/
{alfabeto=["A","B","C","D","E","F","G","H","I","J","K",
  "L","M","N","O","P","Q","R","S","T","U","V","W","X","Y","Z",
   ";",".","'"," "];
}
{da_lettera_a_numero(lettera)=local(j);
     for(j=1,30,if(alfabeto[j]==lettera,return(j-1)));
     error("input non valido.")
}
{da_numero_a_messaggio(num,s="")= local(m,i);
     i=floor(log(n)/log(30))-1;
     while(i>0, s = concat(s,alfabeto[num\(30^i)+1]);
             num = num % (30)^i;i--);
             \\ n \ 30 e' la divisione intera di num per 30
     s = concat(s,alfabeto[num+1]);
     return(s)
}
{da_messaggio_a_numero(w,mod,num=0)= local(l,i,j);
     l=length(w);
     i=floor(log(mod)/log(30));
     M=Vecsmall(w);
          for(j=1,l, num = num +
             30^(i-j)*da_lettera_a_numero(Strchr(M[j])));
```

```
     if(i>l, for(j=l+1,i,num = num +
             30^(i-j)*da_lettera_a_numero(" ")));
         return(num);
}
{genera_chiave_rsa(len,p,q,n,e,d)=
 p=2;q=2;
     until (((isprime(p) && isprime(q))),
               p = nextprime(random(10^(len\2+1)));
               q = nextprime(random(10^(len\2+3)));
     );
     n = p*q; phin = (p-1)*(q-1);
     e = random(phin);
     while(gcd(e,phin)!=1,e=e+1);
     d = lift(Mod(e,phin)^(-1));
     blocco=floor(log(n)/log(30));
     print("La lunghezza massima del blocco codificabile e': ",
               blocco," caratteri");
     return([n,e,d]);
}
{rsa_cifra(messaggio, n, e) = local(l,i,j);
     l=length(messaggio);
     i=floor(log(n)/log(30));
     if (l>i, print("Errore: il messaggio e' di ", l,
          " caratteri ma non deve essere piu' di ", i,
                    " caratteri");
          return);
     lift(Mod(da_messaggio_a_numero(messaggio,n),n)^e);
}
{rsa_decifra(segreto, n, d)  =
             da_numero_a_messaggio(lift(Mod(segreto,n)^d));
}
```

Salvato quanto sopra in un file denominato `rsa.gp` possiamo quindi usufruire di questi programmi per svolgere qualche esempio di codifica e decodifica con RSA.

```
gp > \r rsa
gp > setrand(1) \\ resetta il generatore di numeri casuali
%16 = 1
gp > rsa=genera_chiave_rsa(200)
\\ ritorna [n, e, d], 200 e' il numero di cifre richieste
per n
```

```
La lunghezza massima del blocco codificabile e`: 137 caratteri
%99 = [635559057449398771439982117009717036351708300505 9
775897628703850555062412432868324414420436336943583435343
756735897936754143118096452830820331122689466211435706 0
452245883770311935644336122028491810454286661,

428427147238060736405548986161631227996928483557784263 5
111059898255305928661292291448073056122973580422724152 6
803279243618682296768093840454253385602650754096031162 1
377509854724643306509358084717038988821,

270328031808773797759775221627596259210722368958053829 3
090842135633385953640043028566449700370983017834590994 1
807185185952045344477957742081456602759979036088691378 8
531522411410502205654299829152942410557]


gp >  n = rsa[1]; e = rsa[2]; d = rsa[3];

gp > chiave_pubblica = [n,e]
%101 = [635559057449398771439982117009717036351708300505 9
775897628703850555062412432868324414420436336943583435343
756735897936754143118096452830820331122689466211435706 0
452245883770311935644336122028491810454286661,

428427147238060736405548986161631227996928483557784263 5
111059898255305928661292291448073056122973580422724152 6
803279243618682296768093840454253385602650754096031162 1
377509854724643306509358084717038988821]

gp > msg = "PADOVA"
%37 = "PADOVA"

gp > codifica = rsa_cifra(msg,n,e)
%104 = 300006122044809289652815128974487396825380182577
316786295300122171412262391706641335388010745435349340 9
611980747102020563074123868440515732634493820650847439 8
353934937140525535029678029314284243911312345
```

```
gp >  rsa_decifra(codifica,n,d)
%32 = "PADOVA              "

gp > msg="PROGETTO LAUREE SCIENTIFICHE"
%34 = "PROGETTO LAUREE SCIENTIFICHE"

gp > codifica=rsa_cifra(msg,n,e)
%107 = 10143994644493586434706695344087543382483313413
79930622072355094678250527268616664138068150504541737 9
74161532067038630815264000591434934776784686587947241 1
31618653157601158447476403893347768024189157501 48

gp >rsa_decifra(codifica,n,d)
%36 = "PROGETTO LAUREE SCIENTIFICHE                    "
```

3.6 Appendice: I Quadrati Ripetuti

Un punto che non è strettamente necessario ai nostri Incontri, ma che è interessante, è vedere come si possono calcolare "velocemente" le potenze modulari mediante una strategia detta "metodo dei quadrati ripetuti".

In PARI/Gp è praticamente già implementata questa funzione: basta definire

```
powermod(x,k,m)=lift(Mod(x,m)^k)
```

per poter calcolare $x^k \pmod m$.

Nel caso in cui si avesse a disposizione ancora del tempo o ci fosse qualche studente particolarmente interessato e bravo si potrebbe provare a spiegarlo e vedere se lo studente riesce a scrivere uno script PARI/Gp che lo realizza. Di seguito si fornisce una spiegazione del metodo ed un possibile script.

3.6.1 Il metodo dei quadrati ripetuti per le potenze modulo n

Il problema è calcolare $a^m \pmod n$, dove $m \in \mathbb{N}^*$. Il metodo ingenuo (svolgere $m-1$ prodotti e riduzioni modulo n) è computazionalmente oneroso. È molto più agile scrivere a^m come un prodotto di potenze con base a il cui esponente sia una potenza di 2. Per esempio, per determinare a^{45} basta calcolare a^2, a^4, a^8, a^{16}, a^{32} (cinque elevamenti al quadrato) e poi $a \cdot a^4 \cdot a^8 \cdot a^{32}$, per un totale di sole 8 moltiplicazioni, invece delle 44 necessarie per eseguire il calcolo nel modo consueto. In questo caso l'espansione binaria di 45 è proprio $(45)_2 = 101101$ perché $45 = 1\cdot 2^0 + 0\cdot 2^1 + 1\cdot 2^2 + 1\cdot 2^3 + 0\cdot 2^4 + 1\cdot 2^5 = 1+4+8+32$. Questo ragionamento ci permette di dire che, in generale, sia il numero totale degli elevamenti al quadrato che delle moltiplicazioni è maggiorato dal numero di cifre binarie dell'esponente. Pertanto il caso peggiore si ha quando l'esponente ha tutti 1 nella sua espansione binaria; da quanto sopra è chiaro che si effettuano dunque $2(\lfloor \log_2 m \rfloor + 1) = O(\log m)$ prodotti.

Lo schema dell'algoritmo è:

1) poniamo $P \leftarrow 1$, $M \leftarrow m$, $A \leftarrow a$.

2) Si determinano q ed r rispettivamente quoziente e resto della divisione di M per 2. Se $r = 1$ poniamo $P \leftarrow P \cdot A$.

3) Poniamo $A \leftarrow A^2$, $M \leftarrow q$.

4) Se $M = 0$ l'algoritmo termina e $P = a^m$. Altrimenti si torna al passo 2).

Si veda la anche Figura 3.1. Nel caso $m = 45$ tutti i resti delle varie divisioni, tranne il secondo ed il quinto, sono uguali a 1. Questo algoritmo è particolarmente utile quando si devono fare calcoli modulo un numero molto grande n: facendo seguire

q	r	P	M	A
		1	45	a
22	1	$1\cdot a = a$	22	a^2
11	0	a	11	a^4
5	1	$a\cdot a^4 = a^5$	5	a^8
2	1	$a^5\cdot a^8 = a^{13}$	2	a^{16}
1	0	a^{13}	1	a^{32}
0	1	$a^{13}\cdot a^{32} = a^{45}$	0	

CALCOLO DI POTENZE

1 $P \leftarrow 1$
2 $M \leftarrow m$
3 $A \leftarrow a$
4 **while** $M > 0$ **do**
5 $\quad q \leftarrow \lfloor M/2 \rfloor$ // $q \leftarrow M \gg 1$
6 $\quad r \leftarrow M - 2q$ // $r \leftarrow M\&1$
7 $\quad$ **if** $r = 1$
8 $\quad\quad P \leftarrow P\cdot A$
9 $\quad$ **endif**
10 $\quad M \leftarrow q$
11 $\quad A \leftarrow A^2$
12 **endwhile**
13 **return** P

Figura 3.1: Si osservi che alla fine di ogni ciclo si ha sempre $a^m = P\cdot A^M$.

ad ogni operazione di somma o prodotto il calcolo del resto modulo n, si può fare in modo che tutti i risultati parziali del calcolo siano $\le 2n$. Inoltre, se invece di prendere il minimo resto positivo, si prende il minimo residuo (cioè, se quando il resto $r \in [n/2, n]$ si sceglie $r' := r - n \in [-n/2, 0]$), tutti i risultati parziali dei calcoli sono, in valore assoluto, $\le n$.

Non è difficile scrivere uno script in PARI/Gp che realizzi l'algoritmo dei quadrati ripetuti osservando che, se è già disponibile l'espansione binaria dell'esponente, allora i resti delle divisioni per due sono esattamente le cifre di tale espansione.

In PARI/Gp useremo la funzione `bittest` che consente di estrarre le varie cifre dell'espansione binaria di un intero:

```
gp> ? bittest
bittest(x,n): gives bit number n (coefficient of 2^n) of the
integer x.
```

Ecco uno script PARI/Gp che realizza il metodo dei quadrati ripetuti.

```
 /*********************************************************
*   Quadrati Ripetuti modulo n
*   A. LANGUASCO e A. ZACCAGNINI per il PLS 2005-2006
*********************************************************/

/* input: a,m,n - a intero, m  naturale >=1, n  naturale >=2
```

```
 * output: a^m mod n .
 */

{RepSquares(a,m,n) = local (P, A, L, i);
/*********************************************************
 *   Test sull'input
*********************************************************/
if (((m<=0) || (n<=1)), print("input non valido"));
/*********************************************************
 *   Inizializzazioni
*********************************************************/
P=1;      /* conterra` il risultato parziale  */
A=a;      /* conterra` i vari quadrati  */
i=0;

/*********************************************************
 * calcolo del numero di bit dell'espansione binaria di m
 *********************************************************/
L=floor((log(m)/log(2)))+1;

/*********************************************************/
for (i=0, L-1,
  r= bittest(m, i);  /* r= coefficiente di  2^i
                        nell'espansione binaria di m */
  if(r == 1,    /* se il resto e` 1 moltiplico per
         il risultato parziale del passo
         precedente  */
    P=lift(Mod(P*A,n));
  );
A=lift(Mod(A^2,n));  /* calcolo  il quadrato successivo */
);
/*********************************************************
 *   Stampa del risultato
*********************************************************/
print("a=", a," alla m=", m," modulo n=", n, " e` uguale a ",
           P);
}
```

Ed ecco un esempio di utilizzo di RepSquares (e la verifica con powermod) per il calcolo di $10^5 \pmod 7$.

```
gp> RepSquares(10,5,7)
a=10 alla m=5 modulo n=7 e` uguale a 5
gp> powermod(10,5,7)
%67 = 5
```

BIBLIOGRAFIA

[1] M. Agrawal, N. Kayal, N. Saxena, "PRIMES is in P", *Annals of Mathematics*, 160, pagine 781-793, 2004, disponibile all'indirizzo: `http://annals.math.princeton.edu/2004/160-2/p12`.

[2] H. Cohen, *A Course in Computational Algebraic Number Theory,* Springer-Verlag, Berlin, Heidelberg, New York, terza edizione, 1996.

[3] R. Crandall, C. Pomerance, *Prime numbers. A computational perspective,* Springer-Verlag, Berlin, Heidelberg, New York, seconda edizione, 2005.

[4] W. Diffie, M.E. Hellman, "New directions in cryptography", *IEEE Transactions on Information Theory*, IT-22, pagine 644-654, 1976.

[5] N. Koblitz, *A Course in Number Theory and Cryptography,* Springer-Verlag, Berlin, Heidelberg, New York, seconda edizione, 1994.

[6] A. Languasco, A. Zaccagnini, *Introduzione alla Crittografia,* Ulrico Hoepli Editore, Milano, 2004.

[7] A. Languasco, A. Zaccagnini, *Manuale di Crittografia,* Ulrico Hoepli Editore, Milano, 2015.

[8] A. Languasco, A. Zaccagnini, *Complementi al Manuale di Crittografia,* Ulrico Hoepli Editore, Milano, 2015, `http://www.hoepli.it/editore/hoepli_file/download_pub/978-88-203-6690-2_Complementi.pdf`.

[9] A. Languasco, A. Zaccagnini, "Alcune proprietà dei numeri primi, I", *Sito web Bocconi-Pristem*, 2005, disponibile all'indirizzo: `http://matematica-old.unibocconi.it/LangZac/home.htm`.

[10] A. Languasco, A. Zaccagnini, "Alcune proprietà dei numeri primi, II", *Sito web Bocconi-Pristem*, 2005, disponibile all'indirizzo: `http://matematica-old.unibocconi.it/LangZac/home2.htm`.

[11] A. Languasco, A. Zaccagnini, "Intervalli fra numeri primi consecutivi", *Sito web Bocconi-Pristem*, 2005, disponibile all'indirizzo: `http://matematica-old.unibocconi.it/LangZac/home3.htm`.

[12] S. Singh, *Codici & segreti. La storia affascinante dei messaggi cifrati dall'antico Egitto a Internet.,* Rizzoli, 2001.

[13] A. Zaccagnini, "Cryptographia ad usum Delphini", disponibile all'indirizzo: `http://people.dmi.unipr.it/alessandro.zaccagnini/psfiles/papers/CryptoDelph.pdf`, 2005.

Commenti, critiche, passaggi oscuri, errori di stampa possono essere segnalati agli indirizzi qui sotto.

Prof.	Alessandro Languasco
Indirizzo	Dipartimento di Matematica “Tullio Levi-Civita”, Torre Archimede, via Trieste 63, 35121 Padova
e-mail	alessandro.languasco@unipd.it
pagina web	http://www.math.unipd.it/~languasc

Prof.	Alessandro Zaccagnini
Indirizzo	Dipartimento di Scienze Matematiche, Fisiche e Informatiche, Parco Area delle Scienze, 53/a 43124 Parma
e-mail	alessandro.zaccagnini@unipr.it
pagina web	http://people.dmi.unipr.it/alessandro.zaccagnini/

5 3 ‡ ‡ † 3 0 5)) 6 * ; 4 8 2 6) 4 ‡
.) 4 ‡) ; 8 0 6 * ; 4 8 † 8 ¶ 6 0))
8 5 ; 1 ‡ (; : ‡ * 8 † 8 3 (8 8) 5 *
† ; 4 6 (; 8 8 * 9 6 * ? ; 8) * ‡ (;
4 8 5) ; 5 * † 2 : * ‡ (; 4 9 5 6 * 2
(5 * – 4) 8 ¶ 8 * ; 4 0 6 9 2 8 5) ;
) 6 † 8) 4 ‡ ‡ ; 1 (‡ 9 ; 4 8 0 8 1 ;
8 : 8 ‡ 1 ; 4 8 † 8 5 ; 4) 4 8 5 † 5 2
8 8 0 6 * 8 1 (‡ 9 ; 4 8 ; (8 8 ; 4 (
‡ ? 3 4 ; 4 8) 4 ‡ ; 1 6 1 ; : 1 8 8 ;
‡ ? ;