Chapter 2

Algoritmi aritmetici

2.1 Aritmetica di base

Per gli algoritmi che consideriamo in questo capitolo, il costo computazionale verrà misurato in termini di operazioni sui bit (*bit complexity*). Infatti, in molte applicazioni numeriche di interesse, fra cui le applicazioni crittografiche, la dimensione dei numeri in gioco eccede quella dell'aritmetica di macchina e dunque considerare che il costo delle operazioni sia costante è un'approssimazione decisamente poco accurata.

2.1.1 Addizione, incremento, moltiplicazione e divisione

Addizione e incremento

L'algoritmo di addizione che si impara alle scuole elementari è utilizzabile per numeri rappresentati secondo lo schema posizionale qualunque sia la base di rappresentazione. Nell'esecuzione "manuale" dell'algoritmo, i due numeri devono essere disposti "in colonna" allineati a destra, di modo che le posizioni con lo stesso peso si corrispondano. L'algoritmo procede da destra a sinistra effettuando ad ogni passo l'addizione di al più tre cifre, e cioè due cifre degli addendi più l'eventuale riporto dalla posizione precedente a quella considerata. Chiaramente sulla posizione meno significativa, cioè quella delle unità, le cifre sono solo due perché non c'è riporto. L'addizione di al più tre cifre è dunque l'operazione primitiva utilizzata nell'algoritmo per calcolare il risultato.

Un semplice esempio illustra l'algoritmo nel caso di numeri rappresentati in base 2, che sono particolarmente importanti in Informatica:

$$\begin{array}{rcccccccc} \text{Riporto}: & 1 & 1 & 1 & 1 & 1 & 1 & & \\ & & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ & & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ \hline & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}$$

Le operazioni *primitive* eseguite dall'algoritmo sono l'addizione di due bit, per la posizione meno significativa, e l'addizione di tre bit (i due addendi più il riporto), per tutte le altre. Entrambe queste operazioni producono comunque due valori in uscita, che corrispondono al bit di somma e a quello di riporto. Si noti infatti che, anche nel caso di addizione di tre numeri di un bit, il valore più grande ottenibile è 3, che in base due è rappresentabile con due soli bit: $3_{10} = 11_2$. Questo implica, fra le altre cose, che nell'algoritmo elementare il riporto può solo propagarsi alla posizione immediatamente alla sinistra rispetto a quella in cui si genera.

Le operazioni primitive utilizzate nell'algoritmo elementare di addizione, il cui pseudo-codice è riportato in Figura 2.1, sono tipicamente implementate in hardware usando semplici porte logiche. In particolare sono necessarie porte logiche per le tre funzioni booleane $\wedge$ (*and*), $\vee$ (*or*) e $\oplus$ (*xor*, o *or esclusivo*), la cui definizione ricordiamo per completezza:

$$x \wedge y = \begin{cases} 1 & \text{se } x = y = 1 \\ 0 & \text{altrimenti} \end{cases} \qquad x \vee y = \begin{cases} 0 & \text{se } x = y = 0 \\ 1 & \text{altrimenti} \end{cases}$$

$$x \oplus y = \begin{cases} 1 & \text{se } x \neq y \\ 0 & \text{altrimenti} \end{cases}$$

Utilizzando queste tre funzioni logiche, possiamo esprimere nel modo seguente la somma s e il riporto c dell'addizione di due bit, x e y:

$$\begin{aligned} s &= x \oplus y \\ c &= x \wedge y \end{aligned}$$

Si noti infatti che il bit di somma è 0 se i due addendi sono uguali, mentre è 1 se i due addendi sono diversi. Questo corrisponde precisamente ad effettuare l'or esclusivo dei due addendi. A sua volta il riporto si genera se e solo se i due

```
ADD(x, y)
    // x_{n-1}x_{n-2}...x_1x_0 è la rappresentazione binaria di x
    // y_{n-1}y_{n-2}...y_1y_0 è la rappresentazione binaria di y
1   z_0, c_1 = HALF-ADDER(x_0.y_0)
2   for i = 1 to n - 1
3       z_i, c_{i+1} = FULL-ADDER(x_i, y_i, c_i)
4   return c_n, z_{n-1}, ..., z_1, z_0
```

Figure 2.1: Algoritmo di addizione classico

addendi sono entrambi 1 e dunque la generazione è catturata precisamente dall'and dei due addendi. La primitiva HALF-ADDER utilizzata in Figura 2.1 implementa precisamente questi due semplici calcoli e dunque richiede 2 operazioni sui bit.

L'addizione di tre bit x, y, e z, è leggermente più complessa. Concentriamoci dapprima sul bit di somma. Tale bit è 1 se un solo addendo è 1 oppure se tutti e tre gli addendi sono 1. Questo calcolo può essere realizzato semplicemente mediante l'utilizzo di due porte $\oplus$: infatti vale $(x \oplus y) \oplus z = 1$ precisamente nei due casi appena ricordati. Per quanto riguarda il riporto, l'osservazione fondamentale è la seguente: o il riporto si genera già dalla somma dei primi due bit (e dunque se $x \wedge y = 1$), oppure si genera dalla somma dei primi due con il terzo (e dunque se $(x \oplus y) \wedge z = 1$). Ricapitolando, abbiamo:

$$\begin{aligned} s &= (x \oplus y) \oplus z \\ c &= (x \wedge y) \vee ((x \oplus y) \wedge z) \end{aligned}$$

La primitiva FULL-ADDER utilizzata in Figura 2.1 implementa questi due calcoli con 6 operazioni sui bit.

Prima di studiare il costo computazionale, vogliamo concentrarci su un particolare caso di addizione, che viene utilizzata (spesso "nascostamente") con altissima frequenza negli algoritmi, siano essi numerici o meno. Parliamo dell'operazione di *incremento unitario*. In questo caso il secondo addendo è 1. Questa operazione viene tipicamente eseguita in modo ripetuto, cioè il valore 1 viene ripetutamente sommato al valore di una variabile inizializzata "in qualche modo" (spesso al valore 0 o 1). La riga 2 dell'algoritmo ADD

rappresenta un esempio di tale uso "nascosto" dell'operazione di incremento: la variabile i di controllo dell'iterazione viene qui inizializzata al valore 1 e incrementata di un'unità fino al raggiungimento del valore n.

Se il primo addendo, diciamo x, è un numero di k bit, qual è la complessità del singolo incremento: $x = x + 1$? È facile rendersi conto che, nel caso più sfavorevole, è necessario eseguire esattamente k addizioni di due numeri di un bit (ovvero effettuare k chiamate di HALF-ADDER). Questo chiaramente succede se $x = \underbrace{11\ldots1}_{k \text{ cifre}}$, di modo che ogni addizione genera riporto. La conclusione corretta è dunque che $T_I(k) = O(k)$, dove abbiamo indicato con $T_I(k)$ la complessità di un singolo incremento di una variabile di k bit.

Tuttavia noi siamo interessati non tanto alla complessità di un singolo incremento, bensì a quella di una *sequenza* di n incrementi, perchè questo è ciò che occorre frequentemente negli algoritmi, come mostra proprio il caso del ciclo **for**. Una conclusione corretta, ancorché non adeguata perché eccessivamente pessimistica, si ottiene utilizzando il seguente criterio: il caso più sfavorevole per una seguenza (generica) di passi di calcolo è non peggiore della somma dei casi più sfavorevoli relativi ad ogni singolo passo. Nel nostro casi, poiché abbiamo n incrementi e ogni incremento richiede tempo $O(k)$, l'intera sequenza richiede tempo $O(nk)$. La conclusione (ripetiamo, corretta) è pessimistica perché non è possibile che una sequenza di incrementi della stessa variabile incorra nel worst-case tutte le volte. Vediamo dunque di fare un'analisi più accurata.

Osserviamo, banalmente, che i valori assunti da x a seguito degli n incrementi sono tutti consecutivi. Possiamo quindi utilizzare le seguenti semplici proprietà di immediata verifica:

- un numero binario ogni due consecutivi termina con la cifra 0, e dunque sull'intera sequenza ci sono circa $\frac{n}{2}$ numeri con questa caratteristica;
- un numero ogni quattro termina con le cifre 01;
- ...
- un numero ogni 2^r termina con le cifre $0\underbrace{1\ldots1}_{r-1}$;
- ...

Ora, quando l'incremento viene applicato al (generico) numero che termina con le cifre $0\underbrace{1\ldots1}_{r-1}$, il riporto generato sulla posizione meno significativa si

propaga fino al primo 0 da destra, dopo di che si arresta e non è necessario eseguire altre somme di bit. In questo caso si eseguono quindi r somme di numeri di un bit (dunque implementabili utilizzando la funzione HALF-ADDER).

Supponiamo, per semplicità (ma senza perdita di generalità), che il valore iniziale della variabile x sia 0 e che ad essa vengano applicati n incrementi. Il valore finale sarà ovviamente n, un numero che è rappresentabile con $k = \lceil \log n \rceil$ cifre binarie. Per quanto osservato sopra, il numero totale $T_I(n,k)$ di invocazioni della funzione HALF-ADDER è espresso dalla seguente sommatoria:

$$T_I(n,k) = \sum_{i=1}^{k} \frac{n}{2^i} \cdot i = n \cdot \sum_{i=1}^{k} i \cdot 2^{-i}$$

Si può dimostrare (cf. il Corollario 8.110 in Appendice B) che, per ogni $k \geq 2$, vale $1 \leq \sum_{i=1}^{k} i \cdot 2^{-i} < 2$, per cui risulta:

$$n \leq T_I(n,k) < 2n \tag{2.1}$$

Siamo ora in grado di calcolare la complessità dell'algoritmo di moltiplicazione classico.

Teorema 2.4. Sia $T_{\text{ADD}}(n)$ il tempo impiegato dall'algoritmo ADD su input numeri di n bit. Risulta allora $T_{\text{ADD}}(n) = \Theta(n)$.

Dim. Le istruzioni alle righe 1 e 4 richiedono tempo costante. L'istruzione di riga 3 viene eseguita $n-1$ e ciascuna volta richiede un tempo costante; complessivamente, quindi, il tempo speso su questa riga è $\Theta(n)$. In conseguenza della (2.1) anche il tempo speso su riga 2 è $\Theta(n)$ e dunque effettivamente $T_{\text{ADD}}(n) = \Theta(n)$. □

Si noti che l'algoritmo ADD è ottimale in quanto non è possibile ottenere un tempo sub-lineare (a meno di non usare il parallelismo) in quanto tutti i bit dei numeri in input devono essere acceduti.

Moltiplicazione

Anche l'algoritmo di moltiplicazione elementare può essere applicato a numeri rappresentati in qualunque base, ma risulta particolarmente semplice da descrivere proprio nel caso della base binaria. Si considerino al riguardo due numeri x e y, di n bit ciascuno, di cui si vuole calcolare il prodotto, e si

```
MULTIPLY(x, y)
1  // y_{n-1}y_{n-2}...y_1y_0 è la rappresentazione binaria di y
2  z = 0
3  for i = 0 to n - 1
4       if y_i = 1
5            z = z + x
6       x = x × 2
7  return z
```

Figure 2.2: Algoritmo di moltiplicazione classico

scriva y in binario: $y = y_{n-1}y_{n-2}\ldots y_1y_0$, dove ovviamente y_{n-1} è il bit più significativo. Il valore di tale rappresentazione, nella base 2, è:

$$y = y_{n-1} \cdot 2^{n-1} + y_{n-2} \cdot 2^{n-2} + \ldots + y_1 \cdot 2 + y_0$$

Il prodotto $z = xy$ può quindi essere scritto nel modo seguente, utilizzando la proprietà distributiva:

$$\begin{aligned} z &= x \cdot y \\ &= x \cdot \left(y_{n-1} \cdot 2^{n-1} + y_{n-2} \cdot 2^{n-2} + \ldots + y_1 \cdot 2 + y_0\right) \\ &= y_{n-1} \cdot \left(x \cdot 2^{n-1}\right) + y_{n-2} \cdot \left(x \cdot 2^{n-2}\right) + \ldots + y_1 \cdot (x \cdot 2) + y_0 \cdot x \end{aligned}$$

I generici "prodotti parziali" del tipo $x \cdot 2^i$ possono essere semplicemente ottenuti mediate opportuni shift verso sinistra (di i posizioni) del numero x con l'introduzione di altrettanti 0 a destra. Ad esempio, la rappresentazione di $x \cdot 2^3$ (cioè del numero $8x$) si ottiene mediante shift di tre posizioni verso sinistra della rappresentazione di x. Il prodotto parziale $x \cdot 2^i$ contribuisce poi alla somma finale z se e solo se il corrispondente valore y_i è 1.

Ad esempio, supponiamo che $y = 25_{10} = 11001_2$. In tal caso avremo:

$$z = xy = x \cdot 2^4 + x \cdot 2^3 + x$$

L'algoritmo "completo" è illustrato in figura 2.2. È facile rendersi conto che il suo costo è $O(n^2)$ operazioni sui bit. Infatti, il costo è dominato dalle (al più) n addizioni $z = z + x$, i cui addendi sono numeri di al più $2n$ bit.

```
MULTIPLYR(x, y)
1  if y == 0
2        return 0
3  z = 2 × MULTIPLYR(x, ⌊y/2⌋)
4  if ODD(y)                    // se y è dispari
5        z = z + x
6  return z
```

Figure 2.3: Algoritmo di moltiplicazione ricorsivo

Possiamo chiederci se, come nel caso dell'addizione, anche l'algoritmo elementare di moltiplicazione di figura 2.2 sia ottimale. La risposta questa volta è negativa: esistono algoritmi sensibilmente migliori, come vedremo nel seguito del corso.

Presentiamo ora un secondo algoritmo per la moltiplicazione di interi che, pur avendo anch'esso un costo quadratico, risulta particolarmente interessante come esempio di applicazione di una fondamentale tecnica algoritmica (oltre che una struttura di controllo presente in quasi tutti i linguaggi di programmazione), e cioè la *ricorsione*.

L'algoritmo in questione è una semplice riscrittura (algoritmica, appunto) della seguente identità matematica, che risulta di immediata verifica:

$$x \cdot y = \begin{cases} 2\left(x \cdot \lfloor \frac{y}{2} \rfloor\right) & \text{se } y \text{ è pari} \\ 2\left(x \cdot \lfloor \frac{y}{2} \rfloor\right) + x & \text{se } y \text{ è dispari} \end{cases}$$

L'algoritmo è riportato in figura 2.3.

È facile verificare che anche l'algoritmo MULTIPLYR ha un costo di $O(n^2)$ operazioni sui bit. Infatti, nel caso più sfavorevole, ogni chiamata richiede un'addizione fra numeri di $O(n)$ bit, oltre ad altre operazioni di costo non superiore a $O(n)$. Quindi il costo di ogni chiamata è $O(n)$, e poiché ci sono n chiamate il costo totale è proprio $O(n^2)$.

Divisione

Affrontiamo ora il problema della divisione fra interi. L'algoritmo scolastico in questo caso non ha una formulazione altrettanto semplice come per ad-

```
DIVIDE(x, y)
 1  if x == 0
 2        return (0, 0)
 3  (q̄, r̄) = DIVIDE(⌊x/2⌋, y)
 4  q = 2 × q̄
 5  if EVEN(x)                    // se x è pari
 6        r = 2 × r̄
 7  else r = 2 × r̄ + 1
 8  if r ≥ y
 9        r = r − y
10        q = q + 1
11  return (q, r)
```

Figure 2.4: Algoritmo di divisione ricorsivo

dizione e moltiplicazione; risulta invece molto più agevole l'applicazione della ricorsione.

Ricordiamo che la divisione intera, che indicheremo con l'operatore **div** per distinguerla dalla divisione sui reali, è una funzione che, dati due numeri interi x e $y \neq 0$, restituisce due valori, q ed r, detti rispettivamente *quoziente* e *resto*, i quali soddisfano le seguenti condizioni:

$$x = y \cdot q + r, \qquad 0 \leq r < y$$

Come sappiamo (e come sarebbe facile dimostrare), la condizione $0 \leq r < y$ garantisce l'unicità di q ed r. In particolare vale: $q = \lfloor \frac{x}{y} \rfloor$ e $r = x - \lfloor \frac{x}{y} \rfloor \cdot y$. Il calcolo di quoziente e resto può essere eseguito mediante l'algoritmo ricorsivo, illustrato in figura 2.4, che utilizza uno schema ricorsivo simile a quello della moltiplicazione.

Teorema 2.5. Per ogni coppia di interi positivi x, y tali che $y > 0$, l'algoritmo di figura 2.4 calcola correttamente quoziente e resto della divisione di x per y.

Dim. La dimostrazione procede per induzione. La base è immediata: se $x = 0$ allora vale $x = y \cdot 0 + 0$ ed effettivamente l'algoritmo restituisce la

coppia $(0, 0)$. Per il passo ricorsivo, poniamo:

$$\lfloor \frac{x}{2} \rfloor = y \cdot \overline{q} + \overline{r}$$

ovvero

$$2 \cdot \lfloor \frac{x}{2} \rfloor = 2 \cdot (y \cdot \overline{q} + \overline{r}) = y \cdot (2 \cdot \overline{q}) + 2 \cdot \overline{r}$$

con $\overline{r} < y$. Ora, se x è pari vale $\lfloor \frac{x}{2} \rfloor = \frac{x}{2}$, altrimenti $\lfloor \frac{x}{2} \rfloor = \frac{x-1}{2}$. Ne consegue che, se x è pari la chiamata ricorsiva restituisce una coppia $(\overline{q}, \overline{r})$ che soddisfa l'equazione $x = y \cdot (2 \cdot \overline{q}) + 2 \cdot \overline{r}$, altrimenti essa soddisfa l'equazione $x - 1 = y \cdot (2 \cdot \overline{q}) + 2 \cdot \overline{r}$, ovvero $x = y \cdot (2 \cdot \overline{q}) + 2 \cdot \overline{r} + 1$. Questo giustifica gli assegnamenti nel secondo condizionale dell'algoritmo. Posto quindi

$$q = 2 \cdot \overline{q} \quad \text{e} \quad r = \begin{cases} 2 \cdot \overline{r} & x \text{ pari} \\ 2 \cdot \overline{r} + 1 & x \text{ dispari} \end{cases}$$

come previsto nel codice di figura 2.4, vale $x = y \cdot q + r$. Non abbiamo però ancora certezza che i valori di q ed r siano il quoziente e il resto cercati perché potrebbe risultare $r \geq y$. Tuttavia, poiché $\overline{r} < y$, risulta certamente $r < 2 \cdot y$, anche nel caso in cui x sia dispari. Se dunque $r \geq y$, per ottenere quoziente e resto è sufficiente porre $r = r - y$ e incrementare di un'unità il valore di q. □

Esercizio 2.6. Si dimostri (ma meglio sarebbe dire semplicemente "si verifichi") che il costo computazionale dell'algoritmo di figura 2.4 è $O(n^2)$.

2.2 Aritmetica modulare

Sia m un numero intero positivo e sia x un intero arbitrario. Con la scrittura $x \bmod m$ (letto "x modulo m") si intende il resto della divisione di x per m. Inoltre, con *riduzione modulo* m di un intero x si intende proprio il mapping $x \to (x \bmod m)$.

Parlando di *aritmetica modulare* ci si riferisce genericamente a sistemi di aritmetica degli interi in cui le consuete operazioni interi sono seguite dalla riduzione modulo un intero fissato m, detto appunto *modulo*. Questi sistemi giocano un ruolo fondamentale in molti campi dell'Informatica e, in particolare, come vedremo, nella crittografia.

Iniziamo con l'osservare come l'operazione di riduzione (modulo m) possa essere utilizzata per definire una relazione fra numeri interi:

Definizione 2.7. Due numeri interi x e y sono congruenti (o semplicemente congrui) modulo m se e solo se hanno lo stesso resto nella divisione per m (o, equivalentemente, se la differenza $x - y$ è divisibile per m). Per denotare questo fatto si scrive $x \equiv y \bmod m$.

Si può facilemnte dimostrare che la congruenza è una relazione di equivalenza; essa verifica, cioè, le proprietà riflessiva, simmetrica e transitiva. Fissato m ci sono m classi di equivalenza, una per ogni possibile valore del resto $i = 0, 1, \ldots, m-1$. La i-esima classe è esattamente l'insieme degli interi la cui divisione per m ha resto i.

Esempio 2.8. Le classi di equivalenza dell'insieme dei numeri interi indotte dall'operazione di riduzione modulo 5 sono gli insiemi:

$$\begin{aligned}
\mathcal{C}_0 &= \{0, \pm 5, \pm 10, \pm 15, \ldots\} \\
\mathcal{C}_1 &= \{1, \pm 6, \pm 11, \pm 16, \ldots\} \\
\mathcal{C}_2 &= \{2, \pm 7, \pm 12, \pm 17, \ldots\} \\
\mathcal{C}_3 &= \{3, \pm 8, \pm 13, \pm 18, \ldots\} \\
\mathcal{C}_4 &= \{4, \pm 9, \pm 14, \pm 19, \ldots\}
\end{aligned}$$

La congruenza soddisfa anche la seguente proprietà, detta di *sostituzione*, che è facilmente verificabile usando la definizione.

$$x \equiv \overline{x} \bmod m \quad \text{e} \quad y \equiv \overline{y} \bmod m \quad \Rightarrow \quad \begin{cases} x + y \equiv \overline{x} + \overline{y} \bmod m \\ x \cdot y \equiv \overline{x} \cdot \overline{y} \bmod m \end{cases}$$

Si noti, in particolare, che dato un qualunque intero x e un modulo m, risulta $x \equiv (x \bmod m) \bmod m$, cioè un numero è cogruente modulo m al suo resto modulo m. Quest'ultima semplice osservazione utilizzata con la proprietà di sostituzione permette di effettuare i calcoli in modo molto più efficiente, come mostra il seguente esempio.

Esempio 2.9. In aritmetica modulo 7, supponiamo di dover eseguire il calcolo $(131 \times 226) \bmod 7$. Se procediamo nel modo più immediato calcoliamo dapprima $131 \times 226 = 29606$ dopodiché eseguiamo la divisione intera 29606 **div** 7 ottenendo così il resto 3. La proprietà di sostituzione ci consente invece di eseguire il calcolo nel modo seguente: dapprima $131 \bmod 7 = 5$ e $226 \bmod 7 = 2$, quindi $5 \cdot 2 \bmod 7 = 3$.

In definitiva, fissato il modulo m, è possibile considerare nelle operazioni solo l'insieme di numeri minori di m, indicato con $\mathbb{Z}_m$; tutte le volte che un calcolo intermedio produce un valore maggiore è possibile eseguire la riduzione modulo m in modo da contenere anche significativamente la grandezza dei numeri generati (e quindi anche il tempo di esecuzione). L'esempio più significativo che vedremo al riguardo è quello del calcolo della funzione esponenziale modulare (sezione 2.2.3).

In $\mathbb{Z}_m$ continuano a valere alcune proprietà soddisfatte da addizione e moltiplicazione sugli interi, in particolare:

$$x + (y + z) \equiv (x + y) + z \pmod{m} \qquad \text{Proprietà associativa}$$
$$x \cdot (y + z) \equiv x \cdot y + x \cdot z \pmod{m} \qquad \text{Proprietà distributiva}$$

Oltre ad addizione e moltiplicazione, in $\mathbb{Z}_m$ è sempre definita la sottrazione (qualunque sia il valore di $m > 0$). Questa è una semplice conseguenza del fatto che, per ogni elemento $y \in \mathbb{Z}_m$, esiste l'elemento opposto $-y$, cioè quell'elemento che verifica $y + (-y) \bmod m = 0$. Tale elemento è semplicemente $m - y$: risulta infatti $y + (-y) \bmod m = y + (m - y) \bmod m = m \bmod m = 0$.

Esempio 2.10. L'inverso di 7 in $\mathbb{Z}_9$ è 9-7=2 perchè $2 + 7 \bmod 9 = 0$.

Le operazioni di addizione e moltiplicazione in $\mathbb{Z}_m$ si possono eseguire combinando gli algoritmi visti nella sezione 2.1.1 seguiti dalla divisione (di cui si utilizza solo il resto). Se i numeri in input sono composti di n bit, ricordiamo che l'addizione ha costo $\Theta(n)$ e produce numeri di $n + 1$ bit, mentre la moltiplicazione ha costo $O(n^2)$ e produce numeri di $2n$ bit. La successiva divisione lavora quindi (in entrambi i casi) su numeri di $O(n)$ bit e dunque ha costo $O(n^2)$. Combinando i risultati si vede che sia addizione che moltiplicazione modulare hanno entrambe costo $O(n^2)$ operazioni sui bit.

La divisione modulare non è sempre definita; questo dipende dal fatto che non sempre esiste il reciproco, detto anche inverso moltiplicativo, di un elemento $y \in \mathbb{Z}_m$, cioè quel numero y^{-1} tale che $y \cdot y^{-1} \equiv 1 \bmod m$.

Esempio 2.11. Si consideri l'insieme $\mathbb{Z}_9$. Si può verificare che, per nessun elemento $x \in \mathbb{Z}_9$, vale $3 \cdot x \bmod 9 = 1$. Ne consegue che 3 non ha inverso in $\mathbb{Z}_9$. Se invece consideriamo il numero 4, esso ha un inverso e precisamente 7: vale infatti $4 \cdot 7 \bmod 9 = 28 \bmod 9 = 1$.

Ritorneremo poco più avanti sul problema dell'esistenza dell'inverso e sul suo calcolo effettivo.

2.2.1 Algoritmo di Euclide per il calcolo del MCD

Si tratta di uno dei più antichi algoritmi di cui si abbia conoscenza. Esso è basato sul seguente semplice fatto.

Teorema 2.12. Dati due interi positivi a e b, risulta

$$\text{MCD}(a, b) = \text{MCD}(b, a \bmod b)$$

Dim. Osserviamo intanto che se un numero x divide sia a che b allora divide qualsiasi combinazione lineare di a e b a coefficienti interi: $\alpha \cdot a + \beta \cdot b$, $\alpha, \beta \in \mathbb{Z}$.

Dimostreremo l'asserto facendo vedere che valgono simultaneamente le due disuguaglianze: $\text{MCD}(a, b) \leq \text{MCD}(b, a \bmod b)$ e $\text{MCD}(b, a \bmod b) \leq \text{MCD}(a, b)$.

$\text{MCD}(a, b) \leq \text{MCD}(b, a \bmod b)$**.** Per la proprietà sopra ricordata, $\text{MCD}(a, b)$ divide $a \bmod b$ perché $a \bmod b = a - \lfloor \frac{a}{b} \rfloor \cdot b$ è una combinazione lineare di a e b (con $\alpha = 1$ e $\beta = -\lfloor \frac{a}{b} \rfloor$). Dunque $\text{MCD}(a, b)$ è un divisore comune di b e $a \bmod b$, ma poiché $\text{MCD}(b, a \bmod b)$ è il massimo divisore comune di b e $a \bmod b$, ne consegue che $\text{MCD}(a, b) \leq \text{MCD}(b, a \bmod b)$.

$\text{MCD}(b, a \bmod b) \leq \text{MCD}(a, b)$**.** Analogamente, $\text{MCD}(b, a \bmod b)$ divide b e $a - \lfloor \frac{a}{b} \rfloor \cdot b$ e dunque divide anche la combinazione lineare $1 \cdot \big(a - \lfloor \frac{a}{b} \rfloor \cdot b\big) + \lfloor \frac{a}{b} \rfloor \cdot b = a$. $\text{MCD}(b, a \bmod b)$ è dunque un divisore comune di a e b e dunque non può essere maggiore di $\text{MCD}(a, b)$.

□

L'algoritmo è riportato in figura 2.5. Esso è essenzialmente una implementazione della semplice proprietà appena dimostrata, dalla quale dipende anche la correttezza dell'algoritmo stesso. È però necessario dimostrare che la ricorsione termina e studiare il numero massimo di chiamate ricorsive. Il seguente Lemma consente di determinare la "velocità" di riduzione della grandezza degli operandi nelle varie chiamate ricorsive e dunque di stabilire quante iterazioni (al massimo) consentono di arrivare al caso base terminale.

Lemma 2.13. Se $a \geq b$ allora $a \bmod b < \frac{a}{2}$.

Dim. Consideriamo i due casi: $b \leq \frac{a}{2}$ e $b > \frac{a}{2}$. Il primo caso è banale, perché ovviamente $a \bmod b < b$ ($a \bmod b$ è il resto della divisione di a per b) e dunque $a \bmod b < \frac{a}{2}$. Ma anche l'altro caso è banale, perché se $b > \frac{a}{2}$ allora $\lfloor \frac{a}{b} \rfloor = 1$ e dunque $a \bmod b = a - b \cdot \lfloor \frac{a}{b} \rfloor = a - b < \frac{a}{2}$. □

```
EuclidMCD(a, b)
1  if b == 0
2        return a
3  return EuclidMCD(b, a mod b)
```

Figure 2.5: Algoritmo di Euclide per il calcolo del MCD

Teorema 2.14. L'algoritmo di figura 2.5, su input numeri di n bit termina correttamente in $O(n)$ passi.

Dim. Supponiamo che risulti $a \geq b$, altrimenti il risultato della prima chiamata consiste semplicemente nello scambio degli argomenti. Consideriamo tre chiamate successive:

$$\begin{aligned} \text{EuclidMCD}(a, b) &\rightarrow \text{EuclidMCD}(b, a \bmod b) \\ &\rightarrow \text{EuclidMCD}(a \bmod b, b \bmod (a \bmod b)) \end{aligned}$$

Per il Lemma 2.13 gli argomenti dell'ultima chiamata soddisfano le diseguaglianze $a \bmod b < \frac{a}{2}$ e $b \bmod (a \bmod b) < \frac{b}{2} \leq \frac{a}{2}$. Ne consegue che, ogni due chiamate, il valore degli argomenti quanto meno dimezza e dunque che, dopo al più $2n$ passi, la ricorsione termina. □

Il costo computazionale dell'algoritmo di Euclide è $O(n^3)$ operazioni sui bit, in quanto in ciascuno degli $O(n)$ passi viene eseguita una divisione su numeri di al più n bit.

2.2.2 Divisione modulare

Abbiamo visto che non necessariamente un elemento $x \in \mathbb{Z}_m$ è dotato di inverso moltiplicativo.

Teorema 2.15. Se $\text{MCD}(x, m) > 1$, non esiste inverso moltiplicativo di x modulo m.

Dim. Sia y un generico "candidato" inverso e verifichiamo che necessariamente risulta $x \cdot y \bmod m > 1$. Dalla definizione di modulo abbiamo $x \cdot y \bmod m = x \cdot y - \lfloor \frac{x \cdot y}{m} \rfloor \cdot m$. In altri termini, $x \cdot y \bmod m$ è una combinazione

```
ExtendedEuclid(a, b)
1  if b == 0
2        return (a, 1, 0)
3  (d̄, ā, b̄) = ExtendedEuclid(b, a mod b)
4  return (d̄, b̄, ā − ⌊a/b⌋ · b̄)
```

Figure 2.6: Algoritmo di Euclide esteso

lineare di x ed m. Poiché $\text{MCD}(x, m)$ divide ovviamente sia x che m, esso divide anche tutte le loro combinazioni lineari a coefficienti interi e, dunque, in particolare, divide $x \cdot y \bmod m$. Ma allora $\text{MCD}(x, m) > 1$ implica anche $x \cdot y \bmod m > 1$ e y non è l'inverso di x. □

Che cosa succede, invece, se $\text{MCD}(x, m) = 1$? In tal caso l'inverso esiste e la dimostrazione è costruttiva, nel senso che possiamo esibire un algoritmo che calcola effettivamente l'inverso.

L'algoritmo cui facciamo riferimento è noto come *algoritmo di Euclide esteso*. Su input una coppia di interi x e y, esso produce in uscita tre interi d, a, b, tali che:

$$d = a \cdot x + b \cdot y \quad \text{con} \quad d = \text{MCD}(x, y) \tag{2.2}$$

Se dunque, su input x ed m, l'algoritmo restituisce una terna in cui $d = 1$, allora da $1 = a \cdot x + b \cdot m$ ricaviamo

$$a \cdot x = 1 - b \cdot m$$

e questa relazione dice proprio che $a \cdot x \equiv 1 \bmod m$ e dunque a è l'inverso moltiplicativo di x.

L'algoritmo di Euclide esteso è illustrato in figura 2.6.

Teorema 2.16. L'algoritmo ExtendedEuclid è corretto. Su input $a \geq b \geq 0$ restituisce tre interi d, a, b che soddisfano (2.2).

Dim. La dimostrazione è semplice e procede per induzione. La base è di immediata verifica: se $b = 0$ vale evidentemente $a = 1 \cdot a + 0 \cdot b$ ed inoltre $a = \text{MCD}(a, 0)$. Per il passo induttivo, supponiamo che

$$\overline{d} = \overline{a} \cdot b + \overline{b} \cdot (a \bmod b)$$

ed inoltre che $\overline{d} = \text{MCD}(b, a \bmod b)$. Quest'ultima proprietà, unitamente al Teorema 2.12, garantisce che $\overline{d} = \text{MCD}(a, b)$. Il primo "output" è dunque corretto. Inoltre, ricordando che $a \bmod b = a - \lfloor \frac{a}{b} \rfloor \cdot b$, abbiamo

$$\begin{aligned} \overline{d} &= \overline{a} \cdot b + \overline{b} \cdot (a \bmod b) \\ &= \overline{a} \cdot b + \overline{b} \cdot \left(a - \lfloor \frac{a}{b} \rfloor \cdot b \right) \\ &= \overline{b} \cdot a + \left(\overline{a} - \lfloor \frac{a}{b} \rfloor \cdot \overline{b} \right) \cdot b \end{aligned}$$

che dimostra la correttezza dell'intero output prodotto. □

L'algoritmo di Euclide esteso consente dunque di stabilire se un elemento di $x \in \mathbb{Z}_m$ ha inverso e, in caso affermativo, simultaneamemte anche di calcolarlo utilizzando $O(n^3)$ operazioni aritmetiche, esattamente come l'algoritmo di Euclide "semplice".

Concludiamo questa sezione considerando il caso, molto importante in diverse applicazioni informatiche, in cui il modulo m sia un numero primo. In questo caso, infatti, per quanto appena visto, tutti gli elementi di $\mathbb{Z}_m$ hanno inverso moltiplicativo, ovviamente ad eccezione dello 0. Proprio per escludere lo 0, si definisce anche l'insieme $\mathbb{Z}_m^* = \{1, 2, \ldots, m-1\}$, che, se m è un numero primo, risulta essere un gruppo moltiplicativo. In seguito scriveremo $\mathbb{Z}_p$, o $\mathbb{Z}_p^*$, quando vorremo mettere in evidenza il fatto che il modulo è un numero primo. In $\mathbb{Z}_p^*$ risulta dunque sempre definita anche l'operazione di divisione modulare, nel modo ovvio:

$$x/y \bmod p = x \cdot y^{-1} \bmod p$$

2.2.3 Funzione esponenziale modulare

Per le applicazioni crittografiche, un'altra operazione di grande interesse è la funzione esponenziale: $x^y \bmod m$, dove ovviamente x e y sono numeri interi. In particolare, proprio perché y è intero, il calcolo potrebbe essere effettuato utilizzando semplicemente la definizione:

$$x^y \bmod m = \underbrace{x \cdot x \cdot \ldots \cdot x}_{y \text{ volte}} \bmod m$$

Questo modo di procedere comporterebbe almeno due grossi problemi, il primo dei quali riguarda la dimensione dei numeri.

Supponiamo, al solito, che x, y ed m siano numeri di n bit. Ovviamente anche il risultato finale è un numero di $O(n)$ bit, in quanto è un resto modulo m. Tuttavia, se la riduzione modulo m fosse eseguita solo alla fine, si dovrebbero eseguire calcoli con un numero esponenziale di bit; si noti infatti che x^y è un numero di $\log_2 x^y = y \log_2 x$ bit e la grandezza di y è esponenziale in n. Fortunatamente, grazie alle proprietà dell'aritmetica modulare, le riduzioni si possono effettuare dopo ogni singola operazione senza compromettere la correttezza del calcolo. In questo modo, tutti i risultati intermedi sono numeri di al più $2n$ bit.

Il secondo problema riguarda comunque il tempo di calcolo, perchè come già osservato y è un numero la cui grandezza è $O(2^n)$; dunque eseguire $y - 1$ moltiplicazioni per x richiederebbe tempo esponenziale in n.

Fortunatamente possiamo utilizzare un algoritmo molto più efficiente partendo dalla rappresentazione binaria di y e moltiplicando solo potenze opportune di x. Vediamo prima un esempio.

Esempio 2.17. Sia $y = 13$. Poiché $13_{10} = 1101_2$, cioè $13 = 2^3 + 2^2 + 2^0$, calcoliamo $x^y = x^{2^3+2^2+2^0} = x^{2^3} \cdot x^{2^2} \cdot x^{2^0}$ come prodotto di tre termini di tipo particolare, cioè "x elevato ad una potenza di due". Per il calcolo di queste potenze si può applicare uno schema di "quadratura ricorsiva", calcolando cioè successivamente $x^2 = x \cdot x, x^4 = x^2 \cdot x^2, x^8 = x^4 \cdot x^4$ ed eseguendo quindo solo altre tre moltiplicazioni. In definitiva sono richieste solo 5 moltiplicazioni invece delle 12 dell'algoritmo banale.

L'algoritmo per y generico è mostrato in figura 2.7. Differentemente da quanto visto finora, dell'algoritmo abbiamo dato una formulazione iterativa (anziché ricorsiva). Si noti come vengono "generati", da destra verso sinistra, i bit di y, utilizzando una chiamata dell'algoritmo di divisione. È evidente come lo stesso risultato possa essere ottenuto con semplici operazioni di shift verso destra.

Esercizio 2.18. Scrivere una procedura ricorsiva per il calcolo di $x^y \bmod m$, sul modello dell'algoritmo di moltiplicazione di figura 2.3.

Esercizio 2.19. Dimostrare che la complessità dell'algoritmo di figura 2.7 è $O(n^3)$ operazioni sui bit.

```
ModularExp(x, y, m)
1  p = 1
2  z = x
3  q = y
4  while q > 0
5        (q, r) = divide(q, 2)
6        if r == 1
7              p = p · z mod m
8        z = z · z mod m
9  return p
```

Figure 2.7: Algoritmo di esponenziazione modulare

2.3 Numeri primi

In questa sezione ci occuperemo di test di primalità (dire se un numero è primo o composto), nonché della generazione di numeri primi casuali. Tutto ciò sarà funzionale ad affrontare lo studio (peraltro molto incompleto) dei sistemi crittografici.

2.3.1 Test di primalità

Consideriamo il seguente problema, di semplice enunciazione:

Test di primalità	
Input:	Un numero intero positivo N
Output:	True se N è primo, False altrimenti

Dalle scuole elementari (o al più tardi dalle medie) conosciamo un algoritmo che sembra fare al caso nostro; esso consiste nell'operare la scomposizione in fattori primi: se gli unici fattori che troviamo sono 1 e N, detti anche divisori banali, allora il numero è primo. Bene, abbiamo risolto il problema... Purtroppo no, o meglio, non lo abbiamo risolto con un algoritmo efficiente. Infatti, la scomposizione in fattori vista a scuola prevede di provare in sequenza potenziali divisori via via più grandi: 3, 4, 5, ... Questo può andare bene se si deve effettuare la scomposizione di numeri di piccole

dimensioni. Tuttavia l'algoritmo è esponenziale nel numero di cifre di N, perché il numero di potenziali divisori dipende dalla "grandezza" del numero, che appunto è esponenziale nel numero di bit. Se N ha centinaia di bit, l'approccio è improponibile.

In realtà sappiamo che se un divisore non banale non si trova fra i numeri minori o uguali a $\sqrt{N}$, allora il numero è primo. Questo perché ovviamente non può risultare $N = x \cdot y$ e simultaneamente $x, y > \sqrt{N}$. Limitare la ricerca ai numeri $\leq \sqrt{N}$ è certo un'ottimizzazione, ma non è abbastanza perché $\sqrt{N}$ è comunque un numero di $n/2$ bit, ancora troppo grande considerando ciò che serve nelle applicazioni.

È possibile ridurre ancora, senza grande sforzo, il numero di potenziali divisori da provare. Un esempio sono chiaramente i numeri pari, che possono agevolmente essere esclusi (basta un ciclo for che parte da 3 e ha passo di incremento 2...). Ce ne sono altri che si possono eliminare senza grande sforzo computazionale e sono i multipli di 3 e di 5. Deve però essere semplice individuare se un numero x è multiplo di 3 e/o 5; non è accettabile dividere preliminarmente per 3 (o per 5) e controllare se il resto è 0!

L'algoritmo di Figura 2.8 risolve questo problema organizzando in modo "oculato" la generazione dei candidati. L'algoritmo dapprima prova ad escludere che fra i divisori di N ci siano i numeri primi minori di 10 (2, 3, 5 e 7). Le successive linee di codice si comprendono alla luce della seguente semplice osservazione. Sia x un numero compreso fra 10 e 39. Se x è un multiplo di 2, 3, o 5, allora anche $30 + x$ lo è, perché 30 è il minimo comune multiplo proprio di 2, 3 e 5. Anzi, in generale, ogni numero della forma $x + 30k$ (k intero arbitrario) sarà divisibile per 2, 3 o 5. I numeri che sono divisibili per 2, 3 o 5, nell'intervallo indicato, sono: 10, 12, 14, 15, 16, 18, 20, 21, 22, 24, 25, 26, 27, 28, 30, 32, 33, 34, 35, 36, 38, 39. Quindi non sono da provare ne' questi e neppure tutti i numeri che si ottengono aggiungendo a questi un multiplo qualsiasi di 30.

Il calcolo viene quindi organizzato in base a due cicli: nel ciclo più esterno, a q vengono attribuiti in sequenza i valori $10, 40, 70, \ldots$. A tali valori base vengono aggiunti solo quei piccoli incrementi i tali che $10 + i$ non è incluso nell'elenco dettagliato fornito sopra.

Esercizio 2.20. Come cambierebbe l'algoritmo di figura 2.8 nel caso in cui volessimo eliminare alla radice anche i multipli di 7?

Esercizio 2.21. Come vedremo nella sezione 2.4.2, nella crittografia a chiave pubblica si manipolano numeri composti che sono il prodotto di due soli

```
BRUTEFORCE-PRIMALITY(N)
1  for q ∈ {2, 3, 5, 7}
2       if N mod q = 0
3            return FALSE
4  for q = 10 to √N by 30
5       for i ∈ {1, 3, 7, 9, 13, 19, 21, 27}
6            if N mod (q + i) = 0
7                 return FALSE
8  return TRUE
```

Figure 2.8: Test di primalità di bit complexity esponenziale

primi (scelti a caso) di grandi dimensioni. Supponiamo, come esempio, che $N = p \cdot q$, dove p e q sono numeri primi di 100 bit, di modo che N ha 200 bit. N non è segreto ma è cruciale che tali rimangano p e q. Un potenziale nemico, ben fornito dal punto di vista tencologico ma decisamente sprovveduto da quello algoritmico, decide di usare la procedura per tentativi sopra delineata con l'obiettivo di individuare i fattori p e q. Egli è sicuro che l'aver messo le mani sul supercomputer TIANHE-2 (in grado di eseguire più di $3 \cdot 10^{16}$ operazioni al secondo) gli garantirà il successo. È così sicuro del fatto suo che non si preoccupa neppure di apportare alla procedura le pur semplici migliorie suggerite e semplicemente prova tutti i numeri fra 3 e 2^{100}. Quanti anni impiegherà, il povero diavolo, a trovare i fattori cercati nel caso più sfavorevole?

Esercizio 2.22. Supponiamo che dall'efficienza dell'algoritmo di fattorizzazione appena delineato dipenda la sicurezza con cui la nota *Banca del Buco* difende la segretezza dei conti dei propri clienti. Supponendo che 100 anni siano un orizzonte di sicurezza più che accettabile per la banca (e per i suoi clienti), quale deve essere la lunghezza (in bit) dei numeri primi casuali p e q che la banca sceglie per formare $N = p \cdot q$?

Non sono noti metodi efficienti per fattorizzare un numero, anche se è pur vero che si può fare molto meglio della ricerca esaustiva (si veda oltre in queste note). Questa difficoltà si rivela un prezioso aiuto nella costruzione di protocolli crittografici sicuri. Nell'immediato, però, essa ci impone di tentare

altre strade per stabilire se un dato numero è primo, che è il nostro attuale obiettivo.

Un aiuto sostanziale viene da un teorema di teoria dei numeri, noto come *Piccolo Teorema di Fermat*, per distinguerlo dal più famoso *Ultimo Teorema di Fermat* (dimostrato solo in anni recenti). Il Piccolo Teorema di Fermat, che non dimostreremo, afferma che:

Teorema 2.23. Se N è un numero primo, allora $a^{N-1} \equiv 1 \bmod N$, per ogni $a = 1, 2, 3, \ldots, N-1$.

A prima vista non sembra quel che serve. In input abbiamo N che non sappiamo se è primo e supponiamo di fissare $a = 2$. Se N è davvero primo, calcolando $2^{N-1} \bmod N$ (usando gli algoritmi presentati in questo capitolo) otteniamo ovviamente 1. Però il teorema non dice che, se N non è primo, allora $2^{N-1} \bmod N \neq 1$. Se dicesse anche questo saremmo a posto. Fortunatamente, però, le cose stanno "quasi" in questo modo! Studi sperimentali al calcolatore hanno evidenziato come fra i numeri composti non superiori a $2.5 \cdot 10^{10}$, che sono circa 24 miliardi, ce ne siano solo 20000 che passano il test. Tali numeri vengono anche detti *pseudoprimi base 2*.

Esempio 2.24. 9 è il primo numero dispari composto e vale $2^8 \bmod 9 = 2^4 \cdot 2^4 \bmod 9 = 7 \cdot 7 \bmod 9 = 49 \bmod 9 = 4 \neq 1$. Il successivo numero dispari composto è 15, e risulta $2^{14} \bmod 15 = 2^4 \cdot 2^4 \cdot 2^4 \cdot 2^2 \bmod 15 = 1 \cdot 1 \cdot 1 \cdot 4 \bmod 15 = 4 \neq 1$. Il più piccolo pseudoprimo base 2 è il numero $341 = 13 \cdot 27$. Per calcolare il valore $2^{340} \bmod 341$, è sufficiente osservare che $2^{340} = (2^{10})^{34}$ e che $2^{10} \bmod 341 = 1024 \bmod 341 = 1$. Dunque

$$2^{340} \bmod 341 = (2^{10})^{34} \bmod 341 = 1^{34} \bmod 341 = 1$$

Si noti come abbiamo ripetutamente utilizzato le proprietà dell'aritmetica modulare per rendere efficienti i calcoli eseguiti "a mano".

Nelle moderne applicazioni crittografiche i numeri in gioco sono lunghi centinaia di bit (512 è un valori tipico) e in tale enorme spazio di numeri la presenza dei pseudoprimi base 2 è ancora estremamente più rarefatta. Inoltre, la presenza di numeri che siano pseudoprimi base 2 e simultaneamente pseudoprimi base 3 o base 5 è ancora più rara.

Le considerazioni precedenti consentono di scrivere un algoritmo davvero semplice (figura 2.9) per verificare la primalità di numeri che siano "scelti a

```
PRIMALITY(N)
1  for a ∈ {2, 3, 5}
2        z = a^{N−1} mod N
3        if z > 1
4              return FALSE
5  return TRUE
```

Figure 2.9: Test di primalità per numeri casuali

caso". Quest'ultima precisazione non è per nulla marginale e vedremo subito dopo di approfondirla meglio.

Studiamo prima il costo dell'algoritmo e poi concentriamoci sulla sua "correttezza". Nel caso più sfavorevole, l'algoritmo esegue tre calcoli di potenze modulo N. Se N è un numero di n bit il costo è quindi $O(n^3)$, precisamente il costo dell'esponenziale.

Analizziamo ora la correttezza. Se N è primo, l'algoritmo di figura 2.9 non può commettere errori e restituisce TRUE. Se invece N è composto, allora l'algoritmo può commettere un errore, nel caso in cui N sia simultaneamente pseudoprimo base 2, 3 e 5. Ora, se il numero N è scelto uniformemente a caso fra i numeri di (poniamo) 512 bit, allora è matematicamente possibile parlare di *probabilità di errore* ed affermare che essa è talmente piccola da risultare trascurabile. In effetti, una stima accurata ha portato a valutare che, se anche solo ci si limita a eseguire il test con $a = 2$, tale probabilità (per numeri di 512 bit) è inferiore a 10^{-20}.

Se il numero in input non è stato scelto a caso, allora non si può parlare di probabilità! Se non sappiamo "chi passa gli input al programma" e "come li sceglie" non possiamo escludere che, pur a fronte di una grande rarità degli pseudoprimi, l'input sia sempre proprio uno pseudoprimo! Ripetiamo ancora una volta che, in questo caso, semplicemente non è possibile fare affermazioni sulla probabilità di errore.

Una veloce divagazione. L'algoritmo di figura 2.9 è di tipo decisionale, cioè restituisce in output uno fra due soli possibili valori (TRUE o FALSE). Esso ha anche la particolarità che erra solo se fornisce una delle due possibili risposte, mentre è sempre corretto quando fornisce l'altra. Nel caso particolare, l'algoritmo può errare solo quando dice TRUE (se l'input è in realtà uno

pseudoprimo base 2, 3 e 5). Se invece risponde FALSE, allora è sicuramente corretto. Algoritmi decisionali con questa proprietà vengono etichettati, con terminologia inglese, *one-sided error*.

Ritornando al test di primalità, osserviamo come fortunatamente, per gli scopi crittografici, ciò che serve sono proprio numeri primi scelti a caso e dunque l'algoritmo di figura 2.9 svolge egregiamente il proprio compito. Per concludere questa sezione, osserviamo però come esista un algoritmo di tipo one-sided error, caratterizzato da probabilità di errore trascurabile, che funziona correttamente anche quando l'input non è casuale. In questo caso la casualità è introdotta internamente dall'algoritmo stesso che dunque, oltre ad essere decisionale e one-sided error, è anche classificabile come *algoritmo probabilistico*. Chi fosse interessato ad approfondire questo tema può consultare il testo di Cormen, Leiserson, Rivest e Stein citato nella pagina web del corso, oppure cercare su Internet informazioni sull'algoritmo di *Rabin-Miller*.

2.3.2 Generazione di numeri primi

Non sono noti algoritmi "diretti" per generare numeri primi. Ciò che si può fare è generare numeri casuali della grandezza richiesta, poniamo 512 bit, e poi verificarne la primalità usando il test di figura 2.9, oppure l'ancora più garantista test di Rabin-Miller.

L'algoritmo di generazione, molto semplice da enunciare, visto che il grosso del lavoro è fatto proprio dal test di primalità, è riportato in figura 2.10. Tuttavia è necessario riflettere bene sull'utilizzo della funzione RANDOM (riga 2). Una tale funzione è disponibile nelle librerie di ogni linguaggio di programmazione moderno (e anche meno moderno) e viene utilizzata per produrre sequenze di numeri casuali. L'algoritmo che essa "incorpora" necessita di una sorta di inizializzazione, che può anche essere trasparente all'utente programmatore; dopodiché, ad ogni chiamata, la funzione restituisce un numero diverso.

Il potenziale problema è costituito dal fatto che i numeri restituiti da random non sono affatto casuali, in quanto l'algoritmo di generazione è perfettamente deterministico. Tuttavia, le sequenze di numeri che esso genera "appaiono" casuali ad un osservatore esterno. Di più, essi sono in grado di superare una serie di test statistici volti appunto a verificarne la casualità o comunque "l'aspetto casuale". Per questa ragioni, tali algoritmi vengono comunemente chiamati *generatori pseudocasuali*, così come *pseudocasuali* sono detti i numeri da essi prodotti.

```
PRIMEGEN(N)
1  while true
2       z = RANDOM(N)
3       if PRIMALITY(z) == PRIME
4            return z
```

Figure 2.10: Generazione di un numero primo casuale nell'insieme $\{1, 2, \ldots, N-1\}$.

I numeri pseudocasuali generati dalla funzione di libreria di un linguaggio risultano di sufficiente qualità in moltiplici situazioni applicative. Non è però questo il caso della crittografia, in cui i numeri dovrebbero essere realmente causali. Non possiamo qui addentrarci in una discussione approfondita sul problema della generazione di numeri adatti al caso dei protocolli crittografici. Supporremo semplicemente che la funzione random restituisca numeri realmente casuali, eventualmente con l'aiuto dell'utente, al quale la funzione può "chiedere" interattivamente qualche azione casuale, come ad esempio, muovere ripetutamente il mouse allo scopo di ottenere coordinate casuali sul video da tradurre in numeri[1]).

L'ultima domanda che dobbiamo porci, riguardo l'algoritmo di figura 2.10, riguarda il costo computazionale. Ogni ciclo comporta la generazione di un numero casuale e l'invocazione un test di primalità. Non sappiamo quanto costa la generazione ma possiamo supporre che essa non sia troppo più costosa del test di primalità (nel caso di numeri pseudocasuali, la generazione di un singolo numero richiede l'esecuzione di poche istruzioni aritmetiche). Il vero problema riguarda dunque il numero di tentativi che dobbiamo fare prima di generare davvero un numero primo.

Al riguardo, ci viene in aiuto un noto teorema che afferma che "i numeri primi non sono poi così rari". Dal punto di vista pratico, il teorema garantisce quindi che non dovremo fare troppi tentativi. Più precisamente, indicato con $\pi(N)$ il numero di numeri primi non maggiori di N, il teorema afferma che:

[1]Una tale strategia è utilizzata per generare la chiave privata dal programma `puttygen`, componente di una suite per l'implementazione, in ambiente Windows, delle funzionalità di *secure shell* e *secure file transfer*.

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

dove $\ln N$ indica il logaritmo naturale. Con un po' di libertà, il teorema consente dunque di affermare che, se N è sufficentemente grande, $\pi(N) \approx \frac{N}{\ln N}$ o ancora più utilmente, posto $N = 2^n$

$$\frac{N}{\pi(N)} \approx \ln 2^n = n \ln 2 \approx 0.693n \tag{2.3}$$

La relazione (2.3) è ciò che ci serve dal punto di vista pratico, perchè essa in qualche modo "garantisce" che, per numeri di n bit, si trova un primo ogni poco meno di n numeri. Ad esempio, se consideriamo numeri fino a 512 bit, circa uno su 350 è primo. Da ciò, con passaggi rigorosi, si può dedurre che $O(n)$ generazioni sono sufficienti per imbattersi in un numero primo con alta probabilità. Ricordando la complessità dell'algoritmo di verifica (algoritmo 2.9), questo ci consente di affermare che la complessità dell'algoritmo di generazione è $O(n^4)$ operazioni sui bit. Non poco ma, considerati i valori tipici di n, ben gestibile da un moderno PC.

2.4 Sistemi crittografici

Ci occuperemo ora di un problema cruciale per la sicurezza di molte applicazioni informatiche di fondamentale importanza in ambito civile oltreché (tradizionalmente) militare. Immaginiamo uno scenario in cui i due principali "attori" in gioco vogliono comunicare informazioni riservate attraverso un canale di comunicazione che, come Internet, è altamente insicuro perché accessibile a tutti. Le informazioni, o messaggi, saranno per noi semplicemente sequenze di bit.

Nei testi che trattano di crittografia, ai personaggi in questione vengono ormai tradizionalmente attribuiti i nomi generici di *Alice* (A) e *Bob* (B). Lo scenario (almeno quello che ci interessa in questi appunti) si completa con l'introduzione di altri due attori, uno dei quali è il cattivo di turno, un malintenzionato che cerca di carpire le informazioni riservate. Il nome di *Eve* (E) che si dà a quest'ultimo deriva dal termine *eavesdropper*, che indica una persona che, dall'esterno di una casa, cerca di origliare quello che si dice all'interno. L'ultimo personaggio viene invece indicato con *Trent*,

o semplicemente T, che è l'iniziale di *Trusted*, cioè degno di fede, perché fornisce sempre informazioni corrette sull'identità degli altri attori.

A completamento delle nostre definizioni, indicheremo con M il messaggio "in chiaro" che A e B si vogliono scambiare, con *cifratura* il processo che rende M inintellegibile da parte dei non autorizzati, con C il messaggio cifrato e decifratura il processo che rimette in chiaro il messaggio. Cifratura e decifratura sono formalmente descrivibili in modo funzionale, cioè introducendo due funzioni e e d (*encrypt* e *decrypt*) tali che:

$$C = e(M), \qquad M = d(C)$$

2.4.1 Crittografia a chiave privata

La storia della crittografia per millenni è stata dominata dai cosidetti *protocolli simmetrici*, o a *chiave privata*. In questi schemi, A e B condividono una stessa informazione segreta, che si sono scambiati in modo "sicuro" (ad esempio, incontrandosi di persona) in un momento precedente rispetto al momento dell'uso vero e proprio.

Uno protocollo simmetrico, probabilmente il più semplice da descrivere, è il cosiddetto *one-time pad*, che potremmo tradurre con "il taccuino monuso". In questo schema, A e B condividono una chiave che ha la stessa lunghezza (numero di bit) di M. Indicata con k (*key*) tale sequenza, nel one-time pad le funzioni e e d vengono addirittura a coincidere:

$$e(x) = d(x) = x \oplus k \tag{2.4}$$

dove $\oplus$ indica l'operatore logico "or exclusivo", una funzione booleana che vale True se e soltanto se il valore logico dei due operandi è diverso (l'uno True e l'altro False). Nel caso di sequenze di bit, la (2.4) va intesa come applicata a tutte le coppie di bit corrispondenti. As esempio, nel caso $n = 2$, $01 \oplus 11 = 10$. Il protocollo è dunque specificato dalle seguenti due applicazioni:

$$C = M \oplus k, \qquad M = C \oplus k$$

È infatti immediato verificare che risulta:

$$C \oplus k = (M \oplus k) \oplus k = M \oplus (k \oplus k) = M \oplus \mathbf{0} = M$$

dove **0** indica il vettore di tutti 0. Bob può quindi decifrare il messaggio di Alice semplicemente applicando al testo cifrato la stessa chiave che Alice ha applicato al testo in chiaro in fase di cifratura.

Il protocollo one-time pad è crittograficamente sicuro se:

1. i singoli bit della chiave k sono scelti in modo casuale uniforme (ciascun bit ha la stessa probabilità di essere 0 quanto 1) e indipendente (la scelta del bit i-esimo non influenza quella del bit $i+1$-esimo);

2. ogni singola chiave viene utilizzata una sola volta.

La non ottemperanza a queste direttive rende one-time pad attaccabile. Il requisito 2 spiega anche il nome attribuito al protocollo. Possiamo infatti immaginare che Alice e Bob possiedano due taccuini che riportano una chiave su ogni pagina. Non appena una chiave è stata usata, la pagina corrispondente viene "strappata" in modo da essere sicuri di non poter più riutilizzare la chiave.

In ogni caso, one-time pad rende evidente la criticità del problema dello scambio sicuro di chiavi. Se A e B prevedono di inviarsi n messaggi segreti, devono preliminarmente concordare n chiavi e forse (precauzionalmente) anche di più. Le chiavi devono poi avere sufficiente lunghezza per "coprire" il messaggio più lungo previsto da A e B. In definitiva, il protocollo pone così tanti problemi pratici da essere considerato solo una sorta di esempio giocattolo, utile a spiegare in modo semplice i principi della crittografia a chiave simmetrica.

Ci sono protocolli migliori di one-time pad. Tuttavia è evidente che per le moderne applicazioni informatiche il problema dello scambio sicuro di chiavi, richiesto appunto da tutti i protocolli simmetrici, è davvero il punto critico. Immaginate di voler comprare un articolo in vendita su Amazon, pagando con carta di credito (o in qualunque altro modo; in ogni caso una transazione finanziaria ci deve essere). Le informazioni da tenere riservate, in questo caso, includono chiaramente il numero di carta, il nome del proprietario, la data di scadenza e il codice di sicurezza. Il punto cruciale comunque è che, se fossimo vincolati ad utilizzare un protocollo simmetrico, semplicemente dovremmo concordare con Amazon la chiave! E Amazon dovrebbe concordare chiavi segrete diverse con ciascuno dei suoi clienti (potenzialmente miliardi)! Non è finita: voi dovreste concordare chiavi segrete diverse con qualunque altro rivenditore on-line vogliate fare acquisti! Questa strada è dunque semplicemente impraticabile.

La soluzione al problema precedente viene solo, come vedremo, da un misto di crittografia a chiave pubblica e crittografia a chiave simmetrica. Per concludere, riguardo quest'ultima, descriviamo le caratteristiche comuni a quasi tutti i protocolli moderni, senza entrare nei dettagli.

Un'idea comune è che la chiave non può essere troppo lunga. Valori tipici utilizzati in diversi protocolli sono 128, 192, 256 e 512 bit. La chiave deve poter essere utilizzata più volte su differenti "blocchi" del messaggio che abbiano la stessa lunghezza della chiave (mentre la lunghezza del messaggio è chiaramente arbitraria). Tuttavia, prima di essere applicata al singolo blocco, la chiave viene modificata attraverso una serie di operazioni logico-combinatoriali che devono essere ovviamente reversibili, per garantire la decifratura. Sono proprio queste combinazioni e rendere difficile la decifratura senza la conoscenza della chiave. Va però detto, in conclusione, che anche per i protocolli "approvati" dalle agenzie preposte alla sicurezza, manca una chiara dimostrazione matematica della inviolabilità del codice in assenza della chiave.

2.4.2 Il protocollo RSA

Un protocollo crittografico *a chiave pubblica* risolve alla radice il problema dello scambio di chiavi. In tali sistemi, infatti, non esiste una singola chiave, segreta e condivisa da mittente e destinatario, bensì una coppia di chiavi distinte che sono associate al destinatario (Bob). Una delle due chiavi, che indicheremo con P_B, viene diffusa pubblicamente[2] ed è quella che Alice, o qualunque altro mittente, deve utilizzare per inviare messaggi riservati a Bob. La seconda chiave, detta *privata* perché nota solo a Bob e che indicheremo con S_B, viene utilizzata da quest'ultimo per rimettere in chiaro i messaggi. Un protocollo a chiave pubblica è anche detto *asimmetrico* perché con una singola coppia di chiavi la trasmissione di messaggi riservati può avvenire solo in una direzione (da Alice a Bob). Se il ruolo mittente/destinatario si inverte, allora è necessario che anche Alice si doti di una coppia di proprie chiavi pubblica e privata, P_A e S_A.

La possibilità stessa di realizzare un protocollo a chiave pubblica si basa sulla supposta esistenza di funzioni che sono agevoli da calcolare ma che sono computazionalmente difficili da invertire, in assenza di opportuna informazione segreta. Nel caso dell'RSA la funzione utilizzata è l'esponenziale

[2]Da cui il nome attribuito a questo tipo di protocollo.

modulare. Più precisamente, la trasformazione che viene operata sul messaggio (trattato come numero) è la sequente:

$$C = M^e \bmod N$$

Con opportune scelte di e ed N recuperare M a partire da C (anche conoscendo e ed N) risulta computazionalmente molto costoso, anche se una dimostrazione matematica rigorosa che attesti la complessità del problema non è nota. La decodifica risulta invece semplice se si dispone di un'opportuna informazione; il calcolo che deve essere eseguito è identico a quello della fase di codifica:

$$M = C^d \bmod N$$

e l'informazione segreta è proprio il numero d. Più precisamente e ed N sono la chiave pubblica mentre d è la chiave segreta.

Nel seguito ci sarà utile il seguente fatto, che è una conseguenza immediata della scomposizione in fattori primi di un numero.

Proprietà della fattorizzazione: se un numero X è divisibile per p e per q, allora è divisibile per il minimo comune multiplo di p e q. In particolare, se p e q sono relativamente primi, allora X è divisibile per il prodotto $p \cdot q$.

Procediamo ora con ordine nella spiegazione dei passi necessari per creare N, e e d.

Generazione delle chiavi RSA

1. Generare due numeri primi casuali p e q di sufficiente grandezza (es, 512 bit).
2. Calcolare $N = p \cdot q$.
3. Calcolare la quantità $\Phi(N) = (p-1) \cdot (q-1)$; tale valore coincide con il numero di numeri minori di N e relativamente primi con N.
4. Determinare un numero intero e tale che $\mathrm{MCD}(e, \Phi(N)) = 1$, cioè un numero relativamente primo con $\Phi(N)$.
5. Calcolare d come inverso moltiplicativo di e modulo $\Phi(N)$ usando l'algoritmo di Euclide esteso su input e e $\Phi(N)$. Come sappiamo l'output dell'algoritmo è una terna di numeri m, d, b tali che $m =$

$d \cdot e + b \cdot \Phi(N)$ ed inoltre $m = \text{MCD}(e, \Phi(N)$. Poiché e e $\Phi(N)$ sono relativamente primi, risulta $m = 1$ e quindi

$$d \cdot e = 1 - b \cdot \Phi(N)$$

ma ciò vuol dire esattamente che d è l'inverso moltiplicativo di e modulo $\Phi(N)$.

6. Diffondere $P = (e, N)$ come chiave pubblica e tenere $S = (d, N)$ come chiave segreta.

Esempio 2.25. Posto $p = 29$ e $q = 11$ risulta $N = 319$, $\Phi(N) = 280$. Poiché 280 non è divisibile per 3 possiamo porre $e = 3$, da cui si ricava, usando l'algoritmo Euclide esteso, $d = 187$: risulta infatti $3 \cdot 187 = 561$ e $561 \bmod 280 = 1$. Al messaggio $M = 23$ corrisponde il messaggio cifrato

$$\begin{aligned} C &= 23^3 \bmod 319 \\ &= 529 \cdot 23 \bmod 319 \\ &= 210 \cdot 23 \bmod 319 \\ &= 14 \cdot 15 \cdot 23 \bmod 319 \\ &= 14 \cdot 345 \bmod 319 \\ &= 14 \cdot 26 \bmod 319 \\ &= 364 \bmod 319 \\ &= 45 \end{aligned}$$

Si noti come ancora una volta abbiamo ripetutamente utilizzato le proprietà dell'operatore modulo per ridurre la dimensione dei numeri e rendere più agevoli i conti. Si può poi verificare (anche se "a mano" non sarebbe altrettanto semplice) che $45^{187} \bmod 319 = 23$.

Sono necessarie alcune osservazioni.

Innanzitutto la determinazione di e (passo 3) è fattibile in modo efficiente perché i numeri coprimi con $\Phi(N)$ sono comunque abbondanti. Ad esempio, poiché non è necessario che e sia un numero di grandi dimensioni, si possono provare, in sequenza, i più piccoli numeri primi: 3, 5, 7, 11, ... Se $\Phi(N)$ fosse divisibile per, poniamo, 3, 5 e 7, allora, per la proprietà della fattorizzazione sopra enunciata, $\Phi(N)$ sarebbe anche divisibile

per $3 \cdot 5 \cdot 7 = 105$. Se fosse divisibile pure per 11 e 13, allora lo sarebbe anche per $105 \cdot 11 \cdot 13 = 15015$. Questo "stato di cose" non può andare avanti troppo a lungo perché il prodotto dei (supposti) divisori primi cresce molto velocemente[3] e non può evidentemente eccedere $\Phi(N)$. Esistono peraltro forme di attacco all'RSA che hanno alte probabilità di successo proprio nel caso in cui $\Phi(N)$ sia il prodotto di numeri primi piccoli. Se dunque si dovesse verificare un tale stato di cose, semplicemente di dovrà scartare N e procedere con la scelta di nuovi primi casuali p e q. Per contro, un esponente e piccolo (quando è possibile usarlo senza compromettere la sicurezza) rende computazionalmente più efficiente il processo di cifratura.

Un'altra possibilità è la generazione casuale dell'esponente e. Se si sceglie questa strada i valori "candidati" possono essere interi (dispari) di grandezza arbitraria. Anche con questa scelta, il numero di prove da effettuare prima di trovare un valore relativamente primo con $\Phi(N)$ è limitato. In questo caso, semmai, è importante verificare che d non sia un numero piccolo, perché altrimenti potrebbe essere compromessa la sicurezza. Potrebbe quindi essere preferibile procedere nel modo opposto: per tentativi generare d a caso come numero sufficientemente grande e ottenere e come inverso moltiplicativo modulo $\Phi(N)$.

La seconda osservazione riguarda i passi 4 e 5, che sono stati tenuti logicamente distinti ma che in realtà possono essere fusi in un solo passo. Infatti, fissato un valore e candidato, si applica direttamente l'algoritmo di Euclide esteso alla coppia $(e, \Phi(N))$. Se il valore di $\mathrm{MCD}(e, \Phi(N))$, che è il primo dei tre risultati restituiti dalla funzione, è 1, allora l'inverso d cercato è il secondo risultato restituito. Altrimenti si deve procedere con un nuovo tentativo perché e non ha inverso modulo $\Phi(N)$.

L'ultima osservazione riguarda il ruolo "simmetrico" di e e d. Dal punto di vista puramente matematico il ruolo dei due esponenti può essere rovesciato in quanto:

$$S(P(x)) = (x^e)^d = \left(x^d\right)^e = P(S(x)). \tag{2.5}$$

Fatte salve le questione legate alla sicurezza, di cui abbiamo brevemente parlato sopra, si potrebbe dunque pubblicare d come chiave pubblica e tenere e come chiave segreta. Come vedremo, il ruolo simmetrico giocato dalle chiavi ha però una ben più importante conseguenza; esso rende infatti possibile la realizzazione di protocolli di autenticazione o *firma digitale*.

[3] Il prodotto dei primi 10 numeri primi dispari è un numero di 37 bit, mentre il prodotto dei primi 20 primi dispari ha 95 bit.

Dimostriamo ora la correttezza del protocollo RSA, facendo vedere come il processo di decifrazione eseguito da Bob rimetta effettivamente in chiaro il messaggio.

Teorema 2.26. Il protocollo RSA è corretto, ovvero, se $M < N$ risulta:

$$M = (M^e)^d \bmod N = \left(M^d\right)^e \bmod N$$

Dim. Abbiamo visto che $e \cdot d = 1 + k \cdot \Phi(N)$, per cui possiamo scrivere $(M^e)^d = \left(M^d\right)^e = M^{ed} = M^{1+k(p-1)(q-1)}$. Ora, dimostrare che $M^{1+k(p-1)(q-1)} \bmod N = M$, ovvero che $M^{1+k(p-1)(q-1)} \bmod N - M = 0$, equivale a provare che $M^{1+k(p-1)(q-1)} - M$ è divisibile per N, o ancora (poiché $M < N$), che $M^{k(p-1)(q-1)} - 1$ è divisibile per N.

Se dimostriamo che $M^{k(p-1)(q-1)} - 1$ è divisibile per p e per q, la proprietà della fattorizzazione appena enunciata garantisce che esso è divisibile per $N = p \cdot q$.

Poiché p è primo, il piccolo teorema di Fermat implica che $M^{p-1} \bmod p = 1$ cioè $M^{p-1} = 1 + a \cdot p$ e dunque:

$$M^{k(p-1)(q-1)} - 1 = \left(M^{p-1}\right)^{k(q-1)} - 1 = (1 + a \cdot p)^{k(q-1)} - 1$$

Ora, posto per semplicità $x = k(q-1)$ risulta ovviamente:

$$(1 + a \cdot p)^x = \sum_{i=0}^{x} \binom{x}{i} 1^{x-i} (a \cdot p)^i = 1 + \sum_{i=1}^{x} \binom{x}{i} (a \cdot p)^i$$

e dunque

$$M^{k(p-1)(q-1)} - 1 = \sum_{i=1}^{x} \binom{x}{i} (a \cdot p)^i$$

Poiché in tutti i termini è presente un fattore p, ne consegue che $M^{k(p-1)(q-1)} - 1$ è divisibile per p. Analogamente, si dimostra che $M^{k(p-1)(q-1)} - 1$ è divisibile per q. □

La sicurezza del protocollo RSA si basa sulla difficoltà di calcolare i fattori p e q a partire dalla sola conoscenza di N e di e. Non è infatti noto se esista un modo computazionalmente efficiente per determinare d senza passare dalla fattorizzazione di N, ed ovviamente provare tutti i possibili esponenti è al di fuori di ogni possibilità presente e futura.

Recentemente (2009) è stato sottoposto al tentativo di fattorizzazione un numero di 768 bit (battezzato appunto *RSA-768*). La "sfida" è stata condotta da un team di studiosi appartenenti a più istituzioni di ricerca, utilizzando il miglior algoritmo noto, ovvero il *General Number Field Sieve*[4]. Il tentativo ha avuto successo ma ha richiesto due anni e l'utilizzo di "CPU power" equivalente a circa 2000 anni su un PC moderno. Questi tentativi sono importanti perché permettono di comprendere come si spostano i limiti di sicurezza in funzione del numero di bit dei fattori primi e delle tecnologie (algoritmi, hardware e software) presumibilmente disponibili ai pirati informatici.

2.4.3 Firma digitale

Il fatto che le chiavi pubblica e privata possano essere scambiate dal punto di vista matematico (si ricordi l'equazione (2.5)) ha un'importantissima conseguenza. Supponiamo che Bob voglia inviare un messaggio M ad Alice e che il suo obiettivo non sia tanto di tenere segreto il messaggio, quanto di garantire Alice riguardo l'autenticità del mittente, cioè che è proprio lui che lo ha inviato. Cifrare M con la chiave pubblica di cui Alice si fosse eventualmente dotata non sarebbe una soluzione, perché chiunque potrebbe farlo. L'unica informazione che solo Bob possiede, e che quindi, in qualche modo, lo identifica, è la sua chiave segreta. Supponiamo allora che Bob applichi ad M l'algoritmo RSA utilizzando S_B, e che lo invii ad Alice. Il messaggio F così ottenuto non ha i requisiti di riservatezza, per il semplice motivo che può essere messo in chiaro applicando ad esso nuovamente l'algoritmo RSA con chiave P_B che è nota pubblicamente: $M = P_B(S_B(M)) = P_B(F)$. Tuttavia proprio questo fatto costituisce l'importante conseguenza cui si faceva riferimento: è solo usando la chiave pubblica di Bob che il messaggio viene "decifrato", e ciò costituisce garanzia di autenticità del mittente. Non diremo quindi che il messaggio F è stato cifrato bensì che è stato *firmato digitalmente* da Bob.

In realtà il protocollo è leggermente più elaborato perché quel che Alice ottiene, una volta applicata la chiave pubblica di Bob ad F, è un messaggio M' che è un numero. Ella deve poter avere la certezza che esso coincida con il messaggio/numero M firmato da Bob. Dopotutto, se il messaggio fosse stato

[4]L'algoritmo ha bit complexity, su input numeri di n bit, pari a $e^{(c+o(1))n^{1/3}(\ln n)^{2/3}}$, in cui $c < 2$ e $o(1)$ indica una funzione che tende a zero al crescere di n.

manomesso da Eva, l'applicazione di P_B fornirebbe comunque un numero in output. La soluzione è semplice: Bob invia ad Alice non già $F = S_B(M)$, bensì la coppia $\langle M, F\rangle$, cioè messaggio in chiaro e messaggio firmato. Alice sa che solo nel caso in cui $P_B(F) = M$ può essere certa che il messaggio proviene da Bob e che non è stato manomesso.

Esempio 2.27. Bob vuole inviare il seguente messaggio M ad Alice: "Ti amerò per sempre!". Il messaggio è sufficientemente breve da poter essere riguardato come numero (considerando i bit della codifica ascii dei caratteri giustapposti, spazi inclusi) e poi firmato da Bob, cosicché Alice possa essere certa che è proprio il "suo" Bob che dichiara il proprio amore. Eva, invidiosa perché anch'ella segretamente innamorata di Bob, intercetta il messaggio, o meglio la coppia $\langle M, F = S_B(M)\rangle$. È sua intenzione cambiare M in $\overline{M}$="Amo Eva e fuggirò con lei" o qualcosa di simile. Il problema di Eva è che, per ingannare Alice, non le è sufficiente cambiare M. Ella deve trovare una coppia $\langle \overline{M}, \overline{F}\rangle$ tale che: (1) $\overline{M}$ sia effettivamente un messaggio con il contenuto desiderato, e (2) $P_B(\overline{F}) = \overline{M}$. In caso contrario Alice scoprirebbe che il messaggio è stato manomesso. Non disponendo di S_B, Eva si trova qui nella stessa difficoltà che ha quando vuole mettere in chiaro un messaggio cifrato: o fattorizza N oppure prova tutte le potenziali immagini $\overline{F}$ fino a trovarne una che "funziona", cioè tale che $P_B(\overline{F}) = \overline{M}$. Altre vie non sono note.

È facile rendersi conto di come la firma digitale abbia enormi applicazioni in tutti quei casi in cui è fondamentale stabilire l'autenticità dell'interlocutore. Il commercio elettronico, nell'ambito del quale avvengono transazioni finanziarie, è uno dei campi di applicazione più evidenti, come vedremo nella prossima sezione.

2.4.4 Certificati digitali e autorità di certificazione

Fare acquisti in rete è oggi particolarmente agevole e potenzialmente molto vantaggioso dal punto di vista economico. Bisogna però evitare il rischio di imbattersi negli immancabili truffatori. Un acquisto si conclude sempre con una transazione di denaro e dobbiamo avere la certezza che i versamenti vadano al destinatario legittimo. Dobbiamo cioè sapere con chi stiamo trafficando!

Immaginiamo lo scenario in cui Bob è il venditore e Alice l'acquirente. È quindi Alice che inizia l'interazione con Bob, contattandolo. Se Bob è davvero

Bob bisogna che lo dimostri. Già sappiamo che la crittografia asimmetrica (in particolare RSA) fornisce lo strumento base: Bob firma un messaggio che può essere messo in chiaro solo con la sua chiave pubblica. Il problema, nel campo del commercio elettronico, è che ci sono "troppi Bob" con i quali si può potenzialmente interagire. Se Alice non possiede la chiave pubblica di Bob se la deve procurare, ma chiederla proprio a lui è molto rischioso: Eva potrebbe aver buon gioco nel creare un sito "identico" a quello del negozio virtuale di Bob e far credere ad Alice che è proprio Bob che gli sta inviando la propria chiave pubblica. La soluzione a questo problema è data da una *infrastruttura* tecnologica e non già da un semplice protocollo. Tale infrastruttura è indicata con *PKI* (*Public-Key Infrastructure*).

In una PKI entra in gioco Trent (ed eventualmente più di un personaggio con le caratteristiche di Trent). Questi è un'autorità riconosciuta e degna di fede che conserva un archivio di chiavi pubbliche associate ai loro legali possessori. Più precisamente, per ogni individuo (o ente o azienda) della cui identità Trent si fa garante, egli conserva il cosiddetto *certificato digitale*, cioè una coppia $\langle M, F\rangle$ in cui M include le seguenti informazioni:

- i dati identificativi del soggetto che viene "certificato" (Bob);
- la sua chiave pubblica (di Bob);
- la data di scandenza di validità del certificato;
- i dati identificativi dello stesso Trent.

M viene firmato da Trent con la propria chiave privata; si ha cioè $F = S_T(M)$. La creazione di un certificato per Bob, o meglio, la creazione di una coppia di chiavi publica/privata e la loro attribuzione, univoca e garantita, a Bob, viene fatta una volta per tutte (almeno fino a data di scadenza, quando è necessario un rinnovo): ad esempio Bob va di persona da Trent e questi, verificata l'identità, provvede a creare un certificato che consegna a Bob come sua "carta d'identità digitale": $C_{B,T} = \langle M, F\rangle$ (il doppio indice significa appunto che si tratta del certificato di Bob firmato da Trent). $C_{B,T}$ può essere consegnato ad Alice come a chiunque altro richieda garanzie sull'identità di Bob. Alice, pur non avendo mai contattato Bob in precedenza e dunque non possedendo la sua chiave pubblica, possiede invece la chiave pubblica di Trent. Questo è decisamente più probabile perché le autorità di certificazione sono relativamente poche se confrontate con il numero di potenziali fornitori

di beni e servizi contattabili via Internet. Usando la chiave di Trent su input F, Alice rimette in chiaro M e ne controlla l'integrità, cioè verifica che effettivamente $M = P_T(F)$. A questo punto Alice è sicura che Bob è proprio Bob e può iniziare a mandargli messaggi cifrati con la chiave pubblica P_B rinvenuta nel certificato $C_{B,T}$.

Il quadro si sta dunque facendo più chiaro, ma ci sono ancora due potenziali dubbi che di certo sono balzati alla mente del lettore.

1. È pur sempre possibile che Alice non abbia la chiave pubblica di Trent. Che cosa accade in tale evenienza?

2. Ammesso che ce l'abbia, come ha fatto a procurarsela? È forse dovuta andare "di persona" da Trent?

Alice può benissimo non avere la chiave pubblica di Trent e comunque non è di certo andata da lui a prendersela! Quanti di noi, pur avendo fatto acquisti sicuri in rete, sono mai andati a procurarsi una tale informazione da chicchessia? (Dove sta poi 'sto Trent?)

Il punto di partenza è che Alice ha comunque la chiave pubblica di una qualche *autorità di certificazione*, o semplicemente *CA* (questa è infatti la terminologia giusta per qualificare i personaggi come Trent). Forse non di Trent, ma certamente di qualcuno come lui! In una PKI, tali autorità sono organizzate in maniera gerarchica. Trent, ad esempio, potrebbe essere un'autorità nazionale, la cui identità, con annessa chiave pubblica, è garantita da Tom, un'autorità (pubblica) che ha la responsabilità di tutte le CA nel proprio territorio. A livello internazionale Tom potrebbe poi essere certificato da Ted. Limitando a due, per semplicità, il numero di livelli da "attraversare", supponiamo che Alice non abbia il certificato di Trent ma abbia quello di Tom. In tal caso si può procurare la chiave pubblica di Bob seguendo due volte lo stesso protocollo.

1. Alice contatta Bob, del quale non possiede la chiave pubblica;

2. Bob le dice di essere certificato da Trent;

3. Alice contatta Trent, del quale non possiede la chiave pubblica;

4. Trent le dice di essere certificato da Tom;

5. Alice ha la chiave pubblica di Tom, quindi recupera in modo sicuro la chiave pubblica di Trent e, quindi, quella di Bob.

È chiaro che nessun tipico utente finale implementa direttamente il suddetto protocollo. Tutto ciò avviene in maniera "trasparente"; tipicamente chi implementa i vari passaggi è il browser nel contesto del protocollo http sicuro, o *https*.

Rimane solo da capire come possa Alice avere la chiave pubblica di almeno un'autorità di alto livello senza "dover andare di persona da quest'ultima". La risposta è semplice: quando viene installato un sistema operativo "originale", Alice si ritrova installati anche alcuni certificati validi! Se il mezzo con cui viene installato il software è sicuro (ad esempio, un Windows OEM non "taroccato" oppure distribuito su supporto fisico acquistato in negozio), allora anche i certificati sono autentici.

2.4.5 Sinergia tra crittografia simmetrica e asimmetrica

I protocolli simmetrici si infrangono sul problema dello scambio di chiavi, che invece è risolto, tramite crittografia a chiave pubblica, con le PKI. D'altra parte, il grosso limite dell'RSA è la lunghezza dei messaggi. Usando numeri primi di 512 bit, e dunque un modulo N di 1024 bit, o si tiene il messaggio entro i 1024 bit (perchè il messaggio M, visto come numero, sarebbe chiaramente indistinguibile da ogni altro messaggio $\overline{M} = M + k \cdot N$, $k \in \mathbf{Z}$) oppure si ripete la cifratura su singoli blocchi di 1024 bit, con grande perdita di efficienza.

Una soluzione che viene adottata è allora semplicemente quella di utilizzare la crittografia a chiave pubblica per scambiare in modo sicuro una chiave da utilizzare in un protocollo simmetrico, ad esempio AES. Possiamo quindi completare la descrizione delle attivitè di Alice, iniziata nella precedente sezione, nel modo descritto di seguito.

1. Alice contatta Bob, del quale non possiede la chiave pubblica;
2. Bob le dice di essere certificato da Trent;
3. Alice contatta Trent, del quale non possiede la chiave pubblica;
4. Trent le dice di essere certificato da Tom;
5. Alice ha la chiave pubblica di Tom, quindi recupera in modo sicuro la chiave pubblica di Trent e, quindi, quella di Bob.

6. Alice genera una stringa K di bit casuali (ad esempio, 256 bit) da utilizzare come chiave, detta *chiave di sessione*, in un protocollo simmetrico;

7. Alice cifra la chiave di sessione utilizzando la chiave pubblica di Bob e la invia a quest'ultimo;

8. Bob decifra K;

9. A questo punto Alice e Bob possono iniziare un protocollo di comunicazione simmetrico usando la chiave K nota solo a loro.

Naturalmente ci sono molti dettagli di importanza cruciale per poter raggiungere un sufficiente livello di sicurezza. Tuttavia questo è, a grandi linee, il modo con cui viene implementato il protocollo https.

2.5 Hashing

Questa breve sezione costituisce solo un complemento per (e dunque non sostituisce) la parte relativa all'hashing sul libro di testo. Presentiamo qui semplicemente alcune possibili realizzazioni di funzioni hash di buona qualità utilizzabili per implementare *tabelle hash*.

Nella letteratura sulle strutture dati, le tabelle hash sono una struttura dati concreta utilizzata per realizzare il tipo di dato astratto *dizionario*, ovvero una struttura che supporta almeno le tre operazioni fondamentali di *inserimento*, *ricerca* ed *estrazione* di "oggetti". La natura di tali oggetti dipende della particolare applicazione considerata; è però necessario che essi siano caratterizzati da almeno una proprietà distintiva che viene definita *chiave*. In moltissime applicazioni, e in questi appunti, assumeremo che la chiave sia parte dell'oggetto e che identifichi univocamente l'oggetto stesso. Esempi molto noti di chiavi utilizzate in varie applicazioni gestionali sono: (1) il numero di matricola degli studenti o dei dipendenti di una grande azienda; (2) il codice fiscale dei contribuenti; (3) il codice prodotto, che identifica il tipo di articoli in vendita in un supermercato; (4) le targhe delle automobili registrate al P.R.A.

Una tabella hash è una tabella, dunque una struttura organizzata come sequenza di M posizioni, numerate da 0 a $M-1$, il cui accesso è gestito tramite l'impiego di un'opportuna funzione hash h. Per poter essere utilizzata in tale contesto, la funzione h deve avere come dominio l'universo $\mathcal{U}$ di tutte

le possibili chiavi. Si tratta normalmente di un insieme di grandi dimensioni. Ad esempio, nel caso in cui le chiavi siano codici fiscali, l'insieme universo contiene circa 1.8×10^{17} possibili elementi[5]. Ovviamente, solo un numero molto minore di chiavi vengono concretamente utilizzate nelle applicazioni; tuttavia, la funzione hash deve essere applicabile in linea di principio ad ogni possibile elemento di $\mathcal{U}$.

L'immagine di $\mathcal{U}$ secondo h deve invece essere proprio l'insieme $I = \{0, 1, \ldots, M-1\}$, di modo che i valori calcolati dalla funzione possano essere utilizzati come indici per l'accesso alle corrispondenti posizioni della tabella. Si noti, in generale, che h non può essere iniettiva perché $M < |\mathcal{U}|$, ed anzi spesso $M \ll |\mathcal{U}|$, come dimostra l'esempio dei codici fiscali.

Da un punto di vista strettamente matematico, h non necessita comunque di alcuna altra proprietà. Utilizzando h, la posizione alla quale viene memorizzato un oggetto con chiave k è infatti semplicemente quella individuata dal valore $h(k)$. Questa però è solo l'idea base, che deve essere ulteriormente sviluppata in concreto e resa efficiente da un punto di vista algoritmico.

Una prima obiezione riguarda proprio il fatto che, poiché h non è iniettiva, esistono coppie di potenziali chiavi distinte k_1 e k_2 tali che $h(k_1) = h(k_2)$. Quando si verifica questo fenomeno, al quale viene dato il nome di *collisione*, quale chiave viene memorizzata nella posizione $h(k_1)$? E dove viene memorizzata l'altra (o le altre) con lo stesso valore hash? La gestione delle collisioni è una tecnica squisitamemte algoritmica che caratterizza questa struttura dati.

In letteratura esistono diversi approcci alla gestione delle collisioni. Il più semplice consiste nel memorizzare in ogni posizione j della tabella non già un solo particolare oggetto fra tutti quelli la cui chiave ha come immagine j, bensì il puntatore di testa di una lista (detta appunto *lista di collisione* o anche *lista di trabocco*) nella quale vengono inseriti tutti gli oggetti la cui chiave ha immagine j. La figura 2.5 illustra questa tecnica, chiamata *chaining*.

Supponiamo di utilizzare la tecnica del chaining per risolvere le collisioni e dunque per rispondere alla prima obiezione. In tal caso è però naturale che sorga un'altra perplessità. Viene infatti da chiedersi: è possibile che tutte (o comunque le stragrande maggioranza de) le chiavi utilizzate in una determinata applicazione abbiano la stessa immagine secondo h, e dunque finiscano nella stessa lista di trabocco? Questa sarebbe una situazione alquanto sfor-

[5]Si tratta di un valore approssimato, calcolato sulla base delle regole di formazione dei codici fiscali.

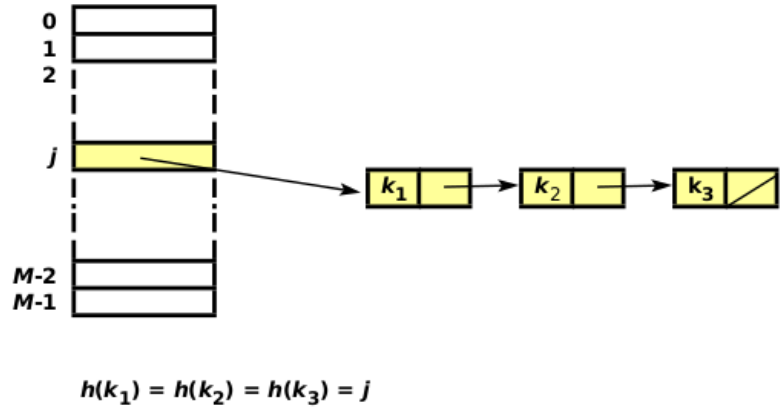

Figure 2.11: Tecnica del chaining per gestire le collisioni. Le tre chiavi (distinte) k_1, k_2 e k_3 hanno la stessa immagine j e quindi vengono inserite nella lista il cui puntatore di testa è memorizzato nella posizione di indice j della tabella. Si noti che l'ordine delle chiavi nella lista dipende da un'ulteriore scelta progettuale (ad esempio, si può decidere di mantenere la lista ordinata per limitare il tempo medio di accesso alle chiavi).

tunata perché in tal caso i tempi di accesso ai singoli oggetti diventerebbero confrontabili con quelli di una semplicissima, ma poco efficiente singola lista concatenata. La risposta all'interrogativo è affermativa: qualunque sia la funzione hash scelta per gestire la tabella, non si può escludere che una tale situazione possa capitare. Tuttavia funzioni hash di buona qualità rendono tale evenienza alquanto remota e vedremo poco oltre in quale modo.

Una ragione banale per cui le liste di trabocco possono diventare lunghe, con conseguenze negative per i tempi di accesso alle chiavi, è l'eventuale sotto-dimensionamento della tabella, cioè del valore M. Osserviamo infatti che se l'applicazione usa n chiavi ci sarà almeno una lista di lunghezza almeno $\lceil n/M \rceil$. Ad esempio, se $n = 10^6$ e $M = 100$, allora almeno una lista ha lunghezza 10000 o superiore. Poiché in molte applicazioni il numero di chiavi effettivamente utilizzate è determinabile con buona approssimazione[6], anche il dimensionamento della tabella può essere spesso fatto con una certa oculatezza. Una regola empirica che fornisce buoni risultati in pratica consiste nel porre $M = 2 \cdot n$. In tal caso la precedente osservazione, ovvero "esi-

[6]Si pensi agli esempi già fatti in precedenza: studenti di una università, dipendenti di un'azienda, contribuenti di un comune, ...

ste almeno una lista di lunghezza almeno $\lceil n/M \rceil$", stabilisce solo un limite inferiore banale perché $\lceil n/M \rceil = \lceil 1/2 \rceil = 1$.

La scelta $M = 2 \cdot n$ non garantisce comunque che non si verifichi il caso patologico in cui tutte le n chiavi abbiano la stessa immagine hash. Questa tutela si ottiene in pratica utilizzando funzioni hash che, pur essendo perfettamente deterministiche, hanno le caratteristiche di funzioni random, anzi (e meglio) funzioni *caotiche*. In modo non rigoroso, definiamo qui caotica una funzione in cui a piccole variazioni nell'input corrispondono in generale grandi variazioni nell'output. Anche se la nozione di vicinanza dovrebbe essere definita in modo rigoroso, preferiamo fare solo qualche esempio pratico che sia illustrativo del concetto. Sono dunque "vicini":

- due codici fiscali che differiscono per due soli caratteri (es., due gemelli di nome Carlo e Carmelo);
- due numeri di matricola consecutivi (attribuiti a due studenti che si trovavano uno dietro l'altro nella fila allo sportello per le immatricolazioni);
- due targhe che differiscono di un solo numero (attribuite ad altrettante macchine vendute nello stesso giorno da uno stesso concessionario).
- due numeri che, quando scritti in binario, differiscono di un solo bit.

Una buona funzione hash applicata agli esempi sopra riportati produrrà valori anche notevolmente differenti. Funzioni hash con queste caratteristiche sono utilizzate ampiamente per scopi crittografici o comunque di sicurezza informatica. Alcuni esempi disponibili in ambiente Unix/Linux sono le funzioni MD5SUM, SHA256SUM e SHA512SUM, che producono sequenze rispettivamente di 128, 256 e 512 bit.

Esempio 2.28. Applicando MD5SUM alle due stringhe `RSSCRL96C14D969W` e `RSSCML96C14D969G`, che rappresentano altrettanti codici fiscali a distanza minima, otteniamo le due sequenze di bit sotto riportate nella più compatta rappresentazione esadecimale:

```
98a14e1be270d3d45679a59bcb47d19c
79a50b299cd0689a152dd6da5e87e17e
```

Come si può notare, le sequenze sono notevolmente "distanti".

Se dunque la dimensione M della tabella è una potenza di 2, cioè $M = 2^k$ per un qualche intero positivo k, allora una buona scelta per la funzione hash consiste nell'utilizzare MD5SUM e considerare solo k bit del risultato (ad esempio, ma non necessariamente, i k bit meno significativi del risultato interpretato come numero). Se M non è una potenza di 2, fissato un valore k tale che $2^k > M$, si possono utilizzare k bit del risultato, visti come numero binario, e operare una successiva riduzione di x modulo M per ottenere un valore nel range desiderato.

Definiamo ora in modo esplicito una *famiglia* di funzioni hash che nel suo insieme gode di un'interessante proprietà, detta di *universalità*. Assumeremo che $\mathcal{U}$ sia l'insieme dei numeri interi di b bit (tipicamente $b = 32$ o $b = 64$).

Per definire la famiglia cominciamo con lo scegliere un modulo M, e dunque anche la dimensione della tabella, che sia un numero primo. In generale questo non è un problema. Infatti i numeri primi sono sufficientemente abbondanti e dunque, qualunque sia la dimensione m inizialmente pensata per la tabella, non è difficile trovare un numero primo $M \geq m$ "vicino" ad m.

Come secondo passo raggruppiamo i b bit delle chiavi in r gruppi di b/r bit ciascuno. Poniamo cioè $k = k_1 k_2 \dots k_r$. Ad esempio, se $b = 64$, potremmo avere 8 gruppi di 8 bit ciascuno oppure 4 gruppi di 16 bit ciascuno. L'unico requisito richiesto nella suddivisione è che $2^{b/r} < M$. Ogni gruppo k_i viene poi visto come un numero intero nell'insieme $\{0, 1, \dots, 2^{b/r} - 1\}$. Concordemente alla suddivione della chiave, consideriamo un vettore a di r numeri interi nell'insieme $\{0, 1, \dots, M - 1\}$:

$$a = (a_1, a_2, \dots, a_r), \qquad a_i \in \{0, 1, \dots, M - 1\}, i = 1, \dots, r$$

Possiamo quindi definire una funzione hash dipendente dal parametro vettoriale a nel modo seguente:

$$h_a(k) = \sum_{i=1}^{r} a_i k_i \bmod M \tag{2.6}$$

Fissato un valore di a, ancora una volta non siamo comunque in grado di escludere che le particolari chiavi utilizzate dall'applicazione (sulla quale, lo ripetiamo ancora una volta, il progettista della struttura dati non ha nessun controllo) abbiano lo stesso valore hash.

Esempio 2.29. Sia $b = 32$, $r = 4$ e $M = 257$. Supponiamo di fissare $a = (125, 174, 83, 211)$. A tutte le seguenti chiavi:

$k_1 = 204895307 = (12, 54, 116, 75)$
$k_2 = 2921059619 = (174, 27, 213, 35)$
$k_3 = 200631156 = (11, 245, 99, 116)$
$k_4 = 1631033338 = (97, 55, 151, 250)$

corrisponde lo stesso valore hash 112.

La dipendenza da un parametro ci consente tuttavia di operare una randomizzazione nell'insieme delle possibili funzioni hash. In altri termini, se proprio non possiamo scegliere le chiavi in modo casuale, sarà la funzione hash ad essere oggetto di scelta randomizzata. Notiamo innanzitutto che, fissato M e fissata la suddivisione in gruppi delle chiavi, esistono M^r possibili scelte del parametro a, questo perché naturalmente ogni a_i può avere uno fra M possibili valori distinti. Dunque esistono altrettante possibili funzioni h_a definite come (2.6). Definiamo quindi:

$$\mathcal{H} = \{h_a | a = (a_1, a_2, \ldots, a_r), a_i \in \{0, \ldots, M-1\}\}$$

La funzione hash utilizzata verrà scelta in modo *casuale uniforme* proprio dalla collezione $\mathcal{H}$. Uniforme vuol dire che ogni funzione ha la stessa probabilità di tutte le altre di essere scelta, ovvero $1/|\mathcal{H}| = 1/M^r$.

Per giungere ad una affermazione precisa dal punto di vista probabilistico è però necessario riflettere sul seguente fatto. La funzione hash viene "cronologicamente" scelta prima delle chiavi. Infatti, la funzione viene decisa in sede di progettazione (e poi di implementazione) della tabella mentre le chiavi vengono presentate durante la fase di utilizzo della tabella. Tuttavia, nell'ipotesi, del tutto realistica, che le particolari chiavi presentate in input non dipendano dalla funzione hash usata[7], è anche possibile immaginare che la situazione sia rovesciata, cioè che prima vengano stabilite tutte le chiavi che verranno utilizzate a tempo di esecuzione (tenendole però opportunamente "nascoste") e dopo venga scelta la funzione hash.

Con questa importante premessa possiamo enunciare il seguente:

[7]Una tale dipendenza fra scelte progettuali e i particolari input presentati ad un algoritmo viene talvolta ammessa per studiare il comportamento worst-case di un algoritmo. In tale scenario si suppone che esista una sorta di *avversario* che, conoscendo i dettagli dell'algoritmo, presenta in input i "peggiori" dati possibili, quelli cioè che lo portano a tempi massimi di esecuzione. Tale studio è spesso funzionale a mostrare come una diversa versione dell'algoritmo, che effettui "con criterio" alcune scelte casuali, renda impossibile all'avversario conoscere quali siano gli input peggiori. Un esempio paradigmatico di questo modo di procedere si vedrà a proposito dello sviluppo della versione randomizzata dell'algoritmo QUICKSORT, nella sezione 3.1.2.

Teorema 2.30. Siano k_1 e k_2 due qualsiasi chiavi distinte. Se la funzione hash h_a è scelta uniformemente a caso nell'insieme $\mathcal{H}$ allora la probabilità che k_1 e k_2 diano origine ad una collisione, cioè che $h_a(k_1) = h_a(k_2)$, è $1/M$.

Dim. Siano $k_1 = (k_1^{(1)}, k_2^{(1)}, \ldots, k_r^{(1)})$ e $k_2 = (k_1^{(2)}, k_2^{(2)}, \ldots, k_r^{(2)})$ le "decomposizioni" di k_1 e k_2 negli r gruppi. Ci chiediamo con quale probabilità risulta $h_a(k_1) = h_a(k_2)$, ovvero

$$\sum_{i=1}^{r} a_i k_i^{(1)} \equiv \sum_{i=1}^{r} a_i k_i^{(2)} \pmod{M} \tag{2.7}$$

Naturalmente, poiché k_1 e k_2 sono distinte, esiste almeno un indice $j \in \{1, \ldots, r\}$, tale che $k_j^{(1)} \neq k_j^{(2)}$. Senza perdita di generalità, ma solo per scrivere le formule in modo più compatto, supponiamo che sia $j = 1$. Possiamo quindi riscrivere la (2.7) nel modo seguente:

$$a_1(k_1^{(2)} - k_1^{(1)}) \equiv \sum_{i=2}^{r} a_i(k_i^{(1)} - k_i^{(2)}) \pmod{M} \tag{2.8}$$

Supponiamo ora di calcolare per primo il valore $\alpha = \sum_{i=2}^{r} a_i(k_i^{(1)} - k_i^{(2)}) \pmod{M}$. Poiché M è un numero primo e la quantità $k_1^{(2)} - k_1^{(1)}$ è non nulla, ne consegue che essa ammette inverso moltiplicativo; esiste cioè il valore $(k_1^{(2)} - k_1^{(1)})^{-1}$. Ma allora la (2.8) sarà soddisfatta se e solo se $a_1 = (k_1^{(2)} - k_1^{(1)})^{-1} \cdot \alpha \pmod{M}$. Poiché i valori a_i sono scelti uniformemente a caso nell'insieme $\{0, 1, \ldots, M-1\}$, ne consegue che a_1 è il valore "giusto" proprio con probabilità $1/M$. □