Algoritmi numerici.

Livio Colussi

14 ottobre 2013

Introduzione

1 Algoritmi aritmetici

Cominceremo con gli algoritmi per eseguire le operazioni aritmetiche. Ci possiamo chiedere se abbia senso in un corso universitario studiare come si eseguono le operazioni aritmetiche visto che il medodo standard per eseguire le operazioni aritmetiche ci è stato insegnato alle scuole elementari. Ma alle scuole elementari eravamo tanto giovani che difficilmente ci siamo chiesti perché quel metodo funziona.

Osserviamo intanto che se vogliamo fare dei conti con i numeri dobbiamo prima scegliere un modo per rappresentarli. Normalmente si usa la rappresentazione decimale (abbiamo dieci dita delle mani per contare): si scelgono dieci simboli $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$ (le dieci cifre decimali); si attribuisce a ciascuno di tali simboli il valore numerico corrispondente ed infine si rappresentano i numeri come stringhe di cifre convenendo che il valore di una stringa di cifre decimali $c_{k-1} \dots c_1 c_0$ è il numero

$$n = c_{k-1} 10^{k-1} + \dots c_1 10 + c_0$$

Naturalmente questo non è l'unico modo per rappresentare i numeri. Invece della base 10 possiamo usare un'altra base. Ad esempio la rappresentazione nella memoria di un computer usa la base binaria: le cifre binarie sono soltanto $0, 1$ (i bit) e il valore di una stringa di bit $b_{k-1} \dots b_1 b_0$ è il numero

$$n = b_{k-1} 2^{k-1} + \dots b_1 2 + b_0$$

Altre basi usate in informatica sono 8 (ottale) e 16 (esadecimale). I Maya usavano la base 20 (non avendo scarpe contavano anche con le dita dei piedi?). [1]

[1] Per essere precisi dovremmo distinguere tra una cifra c (che è soltanto un simbolo) ed il suo valore $\nu(c)$ che è un numero e scrivere $n = \nu(c_{k-1} \dots c_1 c_0) = \nu(c_{k-1}) 10^{k-1} + \dots \nu(c_1) 10 + \nu(c_0)$ per indicare il valore intero n associato alla stringa di cifre $c_{k-1} \dots c_1 c_0$. Ad evitare un eccessivo appesantimento della notazione useremo c sia per indicare la cifra c che il suo valore numerico $\nu(c)$. Il contesto ci dirà se stiamo parlando del simbolo c o del suo valore come numero intero. In particolare gli algoritmi numerici operano sempre su sequenze di cifre che rappresentano numeri e non direttamente su dei numeri (che sono degli oggetti matematici astratti).

Quante cifre servono per rappresentare un numero n in base b? Con k cifre in base b il massimo numero rappresentabile è $b^k - 1$, ad esempio per $k = 3$ e $b = 10$ il massimo numero rappresentabile è $999 = 10^3 - 1$.

Risolvendo rispetto a k otteniamo $k = \lceil \log_b(n+1) \rceil$ e quindi il numero di cifre necessario è compreso tra $\log_b(n) - 1$ e $\log_b(n) + 1$. Dunque $k \approx \log_b(n)$ a meno di una unità.

Come cambia la lunghezza della rappresentazione di un numero al cambiare della base? La formula per il cambiamento di base dei logaritmi è $\log_b n = (\log_a n)/(\log_a b)$ e quindi la lunghezza delle due rappresentazione differisce per il fattore costante $\log_a b$.

Vediamo ora gli algoritmi per le operazioni aritmetiche cominciando dalla somma.

Somma

Osserviamo intanto che, qualsiasi sia la base, il valore della somma di tre cifre si può rappresentare con due sole cifre; ad esempio in base 10 la somma di tre cifre è sempre minore o uguale a $9 + 9 + 9 = 27_{10}$ ed in base 2 la somma di tre cifre è sempre minore o uguale a $1 + 1 + 1 = 11_2$. In generale il valore massimo della somma di tre cifre in base $b \geq 2$ è $3(b-1)$ che si rappresenta con $k = \lceil \log_b(3b-2) \rceil$ cifre. Se $b = 2$ abbiamo $k = \lceil \log_2(4) \rceil = 2$ mentre per $b \geq 3$ abbiamo $k = \lceil \log_b(3b-2) \rceil \leq \lceil \log_b(3b) \rceil \leq \log_b(b^2) = 2$.

Questa semplice proprietà ci permette di sommare due numeri date le loro rappresentazioni in una base qualsiasi. Dobbiamo soltanto allineare a destra le due rappresentazioni e sommare cifra per cifra mantenendo l'overflow come riporto. Poiché ogni singola somma è rappresentabile con due sole cifre il riporto è sempre di una sola cifra e quindi ad ogni passo dobbiamo sommare soltanto tre cifre.

Ecco un esempio in base 2

riporto	1	0	0	1	1	1		
		1	1	0	1	0	1	(53)
		1	0	0	0	1	1	(35)
	1	0	1	1	0	0	0	(88)

L'algoritmo è talmente semplice che non sembra il caso di scriverne lo pseudocodice. Ci limitiamo solo a chiederci quale sia la sua complessità. Poiché ogni passo dell'algoritmo richiede tempo costante e il numero di passi è pari al numero di cifre possiamo dire che per sommare due numeri di k cifre occorre tempo $O(k)$. Possiamo fare meglio? Certamente no: per sommare due numeri occorre almeno leggerli e scrivere il risultato e ciascuna di queste operazioni costa $\Omega(k)$.

Moltiplicazione e divisione

L'algoritmo che abbiamo imparato alla scuola elementare per moltiplicare due numeri interi x ed y consiste nel calcolare una sequenza di addendi ciascuno ottenuto moltiplicando x per una singola cifra di y, incolonnare tali addendi opportunamente spostati a sinistra nella posizione della cifra di y corrispondente e quindi sommare tutti questi addendi così incolonnati. Ad esempio, per moltiplicare 13 per 11, ovvero in notazione binaria $x = 1101$ per $y = 1011$ si procede come segue

			1	1	0	1	$\times$	(rappresentazione binaria di 13)
				1	0	1	=	(rappresentazione binaria di 5)
			1	1	0	1		(1101 per 1)
		0	0	0	0			(1101 per 0, spostato di 1 posto)
	1	1	0	1				(1101 per 1, spostato di 2 posti)
1	0	0	0	0	0	1		(rappresentazione binaria di 65)

In notazione binaria questa operazione è particolarmente semplice in quanto gli addendi intermedi possono essere soltanto 0 e x visto che dobbiamo moltiplicare per una cifra binaria 0 o 1. Osserviamo inoltre che spostare a sinistra di un posto è soltanto un modo semplice per moltiplicare per la base b. La dimostrazione che l'algoritmo è corretto è semplice. Se $y = c_{k-1} \dots c_0$ è la rappresentazione in una qualche base b del numero y allora

$$xy = x \left(\sum_{i=0}^{k-1} c_i b^i \right) = \sum_{i=0}^{k-1} x c_i b^i$$

dove xc_i sono gli addendi ottenuti moltiplicando x per una singola cifra c_i di y e la moltiplicazione per b^i corrisponde allo spostamento a sinistra di i posti di tali addendi.

La complessità di questo algoritmo di moltiplicazione è $\Theta(k^2)$, proporzionale al quadrato della lunghezza k delle rappresentazioni di x ed y.

Il matematico arabo Al Khwarizmi (da cui deriva la parola algoritmo), nel suo trattato di aritmetica scritto circa 11 secoli fa, riporta anche un altro algoritmo per la moltiplicazione: per moltiplicare due interi x e y li si scrive uno accanto all'altro come illustrato nell'esempio seguente e quindi si divide ripetutamente il primo numero per 2 (ignorando il resto se il numero è dispari) e si moltiplica il secondo per 2. Ci si ferma non appena il primo numero diventa 1. In questo modo si ottengono due colonne di numeri. A questo punto si eliminano tutti i numeri nella seconda colonna corrispondenti a numeri pari nella prima colonna. Infine si sommano i numeri rimasti nella seconda colonna ottenendo il risultato voluto.

11	13	
5	26	
2	52	(da eliminare)
1	104	
	143	(risultato)

Lo pseudocodice di questo algoritmo in versione ricorsiva è

MOLTIPLICAZIONE(x, y)

```
1  if x == 0
2        return 0
3  z = MOLTIPLICAZIONE(x, ⌊y/2⌋)
4  if y è pari
5        return 2z
6  else return 2z + x
```

Possiamo vedere questo algoritmo come un esempio della tecnica divide et impera. La sua correttezza deriva dalla relazione di ricorrenza:

$$xy = \begin{cases} 0 & \text{se } y = 0 \\ 2x\lfloor y/2 \rfloor) & \text{se } y \text{ è pari} \\ 2x\lfloor y/2 \rfloor) + x & \text{se } y \text{ è dispari} \end{cases}$$

La complessità di questo agoritmo è data dalla relazione di ricorrenza

$$T(k) = T(k-1) + \Theta(k)$$

la cui soluzione è $T(k) = \Theta(k^2)$, la stessa complessità dell'algoritmo precedente.

Possiamo fare meglio? A prima vista sembrerebbe di no, invece è possibile usando la trasformata veloce di Fourier (FFT). Usando FFT si possono moltiplicare due interi in tempo $\Theta(n \log n)$.

Vediamo ora la divisione. Per dividere l'intero x per un intero $y \neq 0$ bisogna trovare un quoziente q ed un resto r tali che $x = qy + r$ ed $r < y$. Lo pseudocodice di questa operazione è il seguente

DIVISIONE(x, y)

1 **if** x == 0
2 **return** $(q, r) = (0, 0)$
3 $(q, r) =$ DIVISIONE$(\lfloor x/2 \rfloor, y)$
4 $q = 2q$, $r = 2r$
5 **if** x è dispari
6 $r = r + 1$
7 **if** $r \geq y$
8 $r = r - y$, $q = q + 1$
9 **return** (q, r)

Anche questo algoritmo usa la tecnica divide et impera. Lasciamo al lettore dimostrare la correttezza di questo algoritmo e verificare che la complessità è $\Theta(k^2)$, come per la moltiplicazione.

Ci si può chiedere come mai nella normale analisi dei programmi per computer si assume che le operazioni aritmetiche vengano eseguite in tempo costante $\Theta(1)$. In realtà questo è vero soltanto se ci si limita a considerare operazioni aritmetiche su numeri rappresentabili in una parola di memoria RAM. In questo caso il numero di cifre binarie k è limitato da una costante K che rappresenta la lunghezza della parola di memoria (normalmente 32) e quindi $k^2 \leq K^2 = \Theta(1)$. Più in generale possiamo assumere che le operazioni aritmetiche in un algoritmo numerico siano eseguite in tempo costante se è soddisfatta la "condizione della macchina RAM". Tale condizione richiede che le operazioni aritmetiche abbiano operandi rappresentabili con $O(\log |x|)$ cifre binarie dove $|x|$ è il numero di bit necessario a rappresentare l'input x dell'algoritmo.

Per rappresentare numeri con più di 32 cifre binarie occorre usare un certo numero k di parole di memoria e in questo caso possiamo considerare il numero rappresentato con k cifre in base $b = 2^{32}$. In questo caso non possiamo certo assumere che le operazioni aritmetiche si eseguano in tempo costante.

Scomposizione il fattori primi

L'algoritmo più semplice per scomporre in fattori un numero intero n è quello del crivello. Esso consiste nel provare a dividere il numero n per tutti gli interi dispari minori o uguali di $\sqrt{n}$; infatti se n è composto deve avere almeno un fattore minore o uguale a $\sqrt{n}$. La complessità di questo algoritmo è $\Theta(\sqrt{n}k^2) = \Theta(2^{k/2}k^2)$ dove k è il numero di cifre binarie di n.

Troppo per poter scomporre numeri grandi con un migliaio di cifre binarie. Per $k = 1000$ abbiamo infatti $2^{k/2}k^2 = 2^{500}1000^2 \approx 2 \cdot 10^{156}$: servono approssimativamente 10^{140} anni con un moderno supercomputer.

Per il problema della scomposizione in fattori sono stati proposti molti algoritmi sia deterministici che randomizzati. Attualmente l'algoritmo migliore sembra essere quello randomizzato ottenuto con successivi affinamenti dell'algoritmo semplice del crivello []. Benchè la complessità di tale algoritmo randomizzato non sia facilmente determinabile in modo esatto sappiamo che essa cresce esponenzialmente con la radice cubica del numero k di cifre binarie di n. Viene stimato che il tempo di esecuzione medio su un moderno computer sia circa $e^{1.9\sqrt[3]{k}(\ln k)^{2/3}}$ secondi. Per $n = 2^{1000}$ abbiamo $e^{1.9\sqrt[3]{1000}(\ln 1000)^{2/3}} \approx 7 \cdot 10^{28}$ secondi ($2 \cdot 10^{21}$ anni!).

2 Aritmetica modulare

Abbiamo visto che i numeri interi di un computer hanno lunghezza limitata, normalmente 32 cifre binarie. Nella maggior parte delle applicazioni 32 cifre binarie sono più che sufficienti ma per particolari applicazioni, ad esempio per le applicazioni crittografiche, 32 cifre binarie non bastano, ne occorrono alcune centinaia e talvolta migliaia.

L'*aritmetica modulare* ci permette di operare con interi contenuti in un intervallo limitato. Indichiamo con $x \bmod n$ il resto della divisione di x per n; quindi se $x = qn + r$ con $0 \leq r < n$ allora $x \bmod n = r$. Osserviamo che questa definizione di $x \bmod n$ vale anche per gli interi negativi: ad esempio $-13 \bmod 5 = 2$ in quanto $-13 = -3 \cdot 5 + 2$.

Questo introduce una particolare relazione di equivalenza tra tutti gli interi in cui x è equivalente a y modulo n se la differenza $x - y$ è divisibile per n, ovvero in formula

$$x \equiv y \pmod{n} \iff (x - y) \bmod n = 0$$

Possiamo vedere l'aritmetica modulare come un modo per limitare i numeri ad un insieme finito $0, 1, 2, \ldots, n-1$ pensandoli ordinati circolarmente, per cui dopo $n-1$ viene lo 0 (come succede con le ore su di un orologio: quando si arriva alle 24 si riparte da 0).

Alternativamente possiamo vedere l'aritmetica modulare come un modo per raggruppare tutti gli interi in classi di equivalenza. Per ogni i compreso tra 0 ed $n-1$ mettiamo in una stessa classe tutti gli interi del tipo $i + kn$ con k intero qualsiasi. Ad esempio per $n = 3$ abbiamo le tre classi di equivalenza:

$$\begin{array}{cccccccc} \ldots & -9 & -6 & -3 & 0 & 3 & 6 & 9 & \ldots \\ \ldots & -8 & -5 & -2 & 1 & 4 & 7 & 10 & \ldots \\ \ldots & -7 & -4 & -1 & 2 & 5 & 8 & 11 & \ldots \end{array}$$

Rispetto alle operazioni aritmetiche questa equivalenza è una congruenza, nel senso che se prendiamo due interi qualsiasi x ed y e li sommiamo o moltiplichiamo il risultato $x + y$ o xy che otteniamo appartiene alla stessa classe di equivalenza a cui apparterrebbe il risultato ottenuto scegliendo due interi x' ed y' equivalenti rispettivamente a x ed y. Ad esempio con $n = 3$ se prendiamo $x = -5$, $x' = 4$, $y = 8$ e $y' = 5$ abbiamo $x + y = 3$, $x'+y' = 9$, $xy = -40$ e $x'y' = 20$ e, come è facile verificare, $(x+y)-(x'+y') \bmod 3 = 0$ e $xy - x'y' \bmod 3 = 0$.

Formalmente, se $x = i+kn$ e $x' = i+k'n$ appartengono alla stessa classe i e $y = j+hn$ e $y' = j + h'n$ appartengono alla stessa classe j allora

$$(x+y)-(x'+y') \equiv (i+kn+j+hn)-(i+k'n+j+h'n) \equiv (i+j)-(i+j) \equiv 0 \pmod{n}$$

$$xy - x'y' \equiv (i + kn)(j + hn) - (i + k'n)(j + h'n) \equiv (ij) - (ij) \equiv 0 \pmod{n}$$

Vale quindi la seguente

Regola di sostituzione Se $x \equiv y \pmod{n}$ ed $y \equiv y' \pmod{n}$ allora

$$x + y \equiv x' + y' \pmod{n} \quad \text{e} \quad xy \equiv x'y' \pmod{n}$$

Non è difficile verificare che le usuali proprietà di associatività , commutatività e distributività delle operazioni aritmetiche continuano a valere anche in aritmetica modulare:

$$\begin{aligned} x + (y + z) &\equiv (x + y) + z \pmod{n} \\ xy &\equiv yx \pmod{n} \\ x(y + z) &\equiv xy + xz \pmod{n} \end{aligned}$$

Queste proprietà e la regola di sostituzione ci assicurano che quando eseguiamo una sequenza di operazioni aritmetiche possiamo ridurre ad ogni passo i risultati intermedi sostituendoli con il loro resto modulo n. Questa sostituzione può essere estremamente utile quando facciamo calcoli con numeri molto grandi. Ad esempio

$$2^{1315} \equiv (2^5)^{263} \equiv (32)^{263} \equiv 1^{263} \equiv 1 \pmod{31}$$

In aritmetica modulare per sommare due numeri x e y compresi tra 0 ed $n-1$ si effettua la somma solita e quindi si calcola il resto della divisione per n; siccome il risultato è sempre compreso tra 0 e $2n - 2$ per calcolare il resto è sufficiente sottrarre n nel caso in cui il risultato sia maggiore o uguale ad n. Servono quindi una somma, un confronto ed una sottrazione; operazioni che si eseguono in tempo $O(k)$ dove $k = O(\log n)$ è il numero di cifre binarie di x ed y.

Il prodotto di due numeri si effettua allo stesso modo: si calcola dapprima il risultato usando uno dei due algoritmi per la moltiplicazione la cui complessità è $O(k^2)$. Dopo di che si calcola il resto della divisione per n usando l'algoritmo di divisione appena visto. Il risultato xy del prodotto richiede al più $2k$ cifre binarie e dunque la complessità della divisione è anch'essa $O(k^2)$.

La divisione in aritmetica modulare non è altrettanto facile. Nell'aritmetica usuale vi è soltanto un caso eccezionale: la divisione per 0. In aritmetica modulare ci possono essere anche altri casi eccezionali. Riprenderemo in seguito la trattazione della divisione in

aritmetica modulare dopo che avremo visto l'algoritmo di Euclide per il massimo comun divisore.

Per completare le operazioni in aritmetica modulare di cui avremo bisogno consideriamo ora l'elevamento a potenza $x^y \pmod n$. Purtroppo, in questo caso, non possiamo calcolare dapprima x^y e quindi ridurre modulo n come abbiamo fatto per la somma e la moltiplicazione. Il risultato intermedio x^y è troppo grande! Ad esempio, con numeri di 32 cifre binarie x^y possiamo avere $xy = (2^{32}-1)^{2^{32}-1} > (2^{31})^{2^{31}} = 2^{66571993088}$, un numero con più di 66 miliardi di cifre.

Dobbiamo quindi effettuare la riduzione modulo n ad ogni passo intermedio calcolando successivamente

$$x \bmod n, \quad x^2 \bmod n, \quad x^3 \bmod n, \quad \ldots, \quad x^{y-1} \bmod n, \quad x^y \bmod n$$

Ognuna di queste operazioni intermedie richiede tempo $O(k^2)$. Siccome dobbiamo eseguire $y = O(2^k)$ tali operazioni, questo algoritmo richiede un tempo $O(k^2 2^k)$, esponenziale nella dimensione dell'input.

Possiamo fare meglio? Si, basta calcolare dapprima le potenze:

$$x \bmod n, \quad x^2 \bmod n, \quad x^4 \bmod n, \quad x^8 \bmod n, \quad \ldots, \quad x^{2^{k-1}} \bmod n$$

Il calcolo di ciascuno di questi valori richiede soltanto un prodotto per elevare al quadrato il valore precedente. Siccome di questi valori ne dobbiamo calcolare k il tempo totale è $O(k^3)$. Una volta calcolati questi valori ci rimane soltanto da sommare alcuni di essi: quelli che corrispondono alle cifre 1 nella rappresentazione binaria di y. Per l'elevamento a potenza abbiamo quindi un algoritmo di complessità polinomiale $O(k^3)$.

Possiamo riassumere questa idea con lo pseudocodice di un algoritmo ricorsivo che utilizza la seguente regola:

$$x^y = \begin{cases} 1 & \text{se } y = 0 \\ (x^{\lfloor y/2 \rfloor})^2 & \text{se } y \text{ è pari} \\ (x^{\lfloor y/2 \rfloor})^2 x & \text{se } y \text{ è dispari} \end{cases}$$

Lo pseudocodice è il seguente:

```
MODEXP(x, y, n)
1  if y == 0
2       return 1
3  z = MODEXP(x, ⌊y/2⌋, n)
4  if y è pari
5       return z^2 mod n
6  else return (z^2 x) mod n
```

3 L'algoritmo di Euclide per il MCD

L'algoritmo di Euclide, scoperto più di 2000 anni fa dal matematico greco Euclide, calcola il massimo comun divisore (MCD) di due interi x ed y ossia il massimo intero che li

divide entrambi. Siccome ogni intero divide 0 il MCD è definito soltanto se x ed y non sono entrambi nulli. L'algoritmo di Euclide si basa sull'osservazione che $MCD(x, y) = MCD(y, x)$ e $MCD(x, y) = MCD(-x, y)$ (i divisori comuni di x ed y sono ovviamente anche divisori comuni di y ed x e di $-x$ e y), che $MCD(x, 0) = |x|$ e che $MCD(x, y) = MCD(x - y, y)$ (ogni intero che divide x ed y divide anche $x - y$ e viceversa ogni intero che divide $x - y$ e y deve dividere anche x). Per le prime due proprietà possiamo assumere che x ed y siano entrambi non negativi e che $x \geq y$. In questa situazione, sapendo che x ed y non sono entrambi nulli, possiamo usare ripetutamente la terza proprietà per ottenere $MCD(x, y) = MCD(x \bmod y, y)$ e quindi la prima per ottenere $MCD(x, y) = MCD(y, x \bmod y)$. L'algoritmo di Euclide è quindi il seguente

MCD(x, y) // $x \geq y$ non negativi e non entrambi nulli.

```
1  if y == 0
2       return x
3  else d = MCD(y, x mod y)
4       return d
```

La correttezza di questo algoritmo deriva immediatamente dalle proprietà del MCD che abbiamo appena visto.

Per la complessità cominciamo con osservare che se $x \geq y$ allora $x \bmod y \leq x/2$: infatti, se $y \leq x/2$ allora $x \bmod y < y \leq x/2$ mentre se $y > x/2$ allora $x \bmod y \leq x - y < x/2$. Questo significa che la rappresentazione binaria dell'argomento $x \bmod y$ della chiamata ricorsiva ha almeno una cifra in meno della rappresentazione binaria di x. Dunque ad ogni chiamata ricorsiva il numero totale di cifre binarie necessarie per rappresentare l'input (i due parametri x ed y) diminuisce almeno di una unità. Questo significa che se i valori iniziali di x ed y sono numeri di k cifre binarie il numero di chiamate ricorsive è al più $2k$. Siccome ogni chiamata ricorsiva richiede una divisione che ha complessità $O(k^2)$, la complessità dell'algoritmo di euclide è $O(k^3)$.

A questo punto possiamo riprendere la discussione sulla divisione in aritmetica modulare. Per fare questo consideriamo la seguente versione aumentata dell'algoritmo di Euclide.

MCD-AUM(x, y) // $x \geq y$ non negativi e non entrambi nulli.

```
1  if y == 0
2       return (x, 1, 0)
3  else (d, a', b') = MCD-AUM(y, x mod y)
4       return (d, b', a' - ⌊x/y⌋b')
```

Questo algoritmo ritorna una terna di interi (d, a, b) invece di un solo valore. Il primo dei tre valori d è il MCD di x ed y mentre a e b sono due interi tali che $d = ax + by$.

Che MCD-AUM calcoli $d = MCD(x, y)$ è ovvio in quanto lo calcola allo stesso modo di MCD. Rimane da dimostrare che esso calcola a e b tali che $d = ax + by$. Dato che l'algoritmo è ricorsivo usiamo la dimostrazione per induzione sul valore del secondo argomento y.

Il caso base $y = 0$ è banale: $d = x = 1 \cdot x + 0 \cdot y = ax + by$.

Per $y > 0$ l'algoritmo calcola dapprima $(d, a', b') = \text{MCD-AUM}(y, x \bmod y)$. Siccome $x \bmod y < y$ possiamo assumere che la chiamata ricorsiva calcoli correttamente (d, a', b') tali che $d = MCD(y, x \bmod y) = MCD(x, y)$ e che $d = a'y + b'(x \bmod y)$. Esprimendo il resto $x \bmod y$ come $x - \lfloor x/y \rfloor y$ otteniamo

$$d = MCD(x, y) = a'y + b'(x - \lfloor x/y \rfloor y) = b'x + (a' - \lfloor x/y \rfloor b')y$$

il che dimostra che il risultato $(d, b', a' - \lfloor x/y \rfloor b')$ ritornato dall'algoritmo è corretto.

La complessità dell'algoritmo esteso di Euclide rimane $O(k^3)$.

Divisione in aritmetica modulare

Nei numeri reali ogni numero $x \neq 0$ ha un inverso $x^{-1} = 1/x$ e dividere per x è lo stesso che moltiplicare per x^{-1}. In aritmetica modulare daremo una definizione simile di divisione.

Diremo che a è un *inverso moltiplicativo modulo* n di x se $ax \equiv 1 \pmod{n}$.

Purtroppo un tale inverso moltiplicativo non sempre esiste. Ad esempio 2 non ha inverso moltiplicativo modulo 6: infatti $2a \not\equiv 1 \pmod{6}$ qualunque sia a in quanto $2a - 1$ è dispari e quindi non può essere divisibile per 6. Più in generale, se $MCD(x, n) > 1$ allora $ax - 1$ non è divisibile per $MCD(x, n)$ e quindi neppure per n.

Questo è il solo caso in cui x non ha inverso moltiplicativo. Se $MCD(x, n) = 1$, nel qual caso diremo che x ed n sono relativamente primi, allora l'algoritmo esteso di Euclide fornisce due valori a e b tali che $ax + bn = MCD(x, n) = 1$ il che comporta $ax \equiv 1 \pmod{n}$. Quindi a è un inverso moltiplicativo di x.

Dunque, se $MCD(x, n) = 1$ esiste un inverso a modulo n di x e, usando l'algoritmo esteso di Euclide, possiamo calcolarlo in tempo $O(k^3)$ dove k è il numero di cifre binarie di n. La successiva moltiplicazione per $a = x^{-1}$ si esegue in tempo $O(k^2)$ e quindi la divisione in aritmetica modulare si esegue in tempo $O(k^3)$.

4 Il test di primalità

Il più semplice test di primalità di un numero intero n consiste nel provare a dividere n per tutti gli interi dispari minori di $\sqrt{n}$. Purtroppo questo test ha complessità $\Theta(\sqrt{n})$, troppo per poterlo eseguire in modo efficiente con numeri anche soltanto di 256 cifre binarie (per effettuare le 2^{127} divisioni necessarie servono circa 10^{22} anni su un supercomputer che esegue 10^9 divisioni al secondo).

Per effettuare il test di primalità in modo efficiente ci baseremo sul seguente risultato ottenuto da Fermat nel 1640:

Teorema 4.1 (Piccolo teorema di Fermat) *Se p è primo allora per ogni $1 \leq a < p$*

$$a^{p-1} \bmod p = 1$$

Dimostrazione. Sia $S = \{1, 2, \ldots, p-1\}$ l'insieme di tutti gli interi positivi minori di p e sia $1 \leq a < p$. Chiaramente $xa \bmod p \neq 0$ per ogni $x \in S$ in quanto x e a non sono divisibili per p e p è primo. Inoltre se $x > y$ sono due elementi distinti di S allora

$xa \bmod p \neq ya \bmod p$ in quanto la differenza $xa - ya = (x - y)a$ non è divisibile per p. Dunque gli interi

$$1 \cdot a \bmod p \qquad 2 \cdot a \bmod p \quad \ldots\ldots \quad (p-1) \cdot a \bmod p$$

sono tutti distinti e diversi da 0 e sono quindi una permutazione degli elementi di S. Se li moltiplichiamo tra loro otteniamo

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

e dividendo per $(p-1)!$ (cosa che possiamo fare in quanto $(p-1)!$ è relativamente primo con p) otteniamo $a^{p-1} \bmod p = 1$. □

Per effettuare il test di primalità usando il teorema precedente non basta verificare $a^{p-1} \bmod p = 1$ per qualche valore di a; per certi valori di a può succedere che $a^{p-1} \bmod p = 1$ anche se p non è primo (ad esempio $341 = 11 \cdot 31$ non è primo anche se $2^{340} \bmod 341 = 1$).

Occorre quindi verificare che $a^{p-1} \bmod p = 1$ per ogni $1 \leq a < p$ e questo richiede $\Theta(n)$ divisioni! Troppe per un test efficiente. Inoltre il piccolo teorema di Fermat non dice cosa succede se p non è primo. Esistono infatti alcuni numeri composti per i quali, come per i numeri primi, succede che $a^{p-1} \bmod p = 1$ per ogni $1 \leq a < p$. Questi numeri sono detti *numeri di Carmichael*. Il più piccolo numeri di Carmichael è $561 = 3 \cdot 11 \cdot 13$ che è composto ma $a^{560} \equiv 1 \pmod{561}$. Per fortuna i numeri di Carmichael sono estremamente rari. Per il momento ignoriamo l'esistenza dei numeri di Carmichael e supponiamo di lavorare in un mondo privo di tali numeri. Vedremo in seguito come trattare tali numeri.

Possiamo scegliere casualmente un numero $a < p$ e controllare se $a^{p-1} \bmod p = 1$. Se la risposta è no siamo sicuri che p non è primo, se la risposta è si sappiamo soltanto che esso può essere primo ma non ne abbiamo la certezza. Avendo scelto casualmente a possiamo chiederci quale sia la probabilità che la risposta sia si anche se il numero p non è primo. Per calcolare questa probabilità useremo il seguente risultato

Lemma 4.2 *Se $a^{n-1} \not\equiv 1 \pmod{n}$ per qualche a relativamente primo con n allora questo deve valere almeno per la metà delle possibili scelte di a.*

Dimostrazione. Sia a un intero relativamente primo con n per cui $a^{n-1} \not\equiv 1 \pmod{n}$. Ogni b per cui vale $b^{n-1} \equiv 1 \pmod{n}$ ha un gemello $c = ab$ tale che $c^{n-1} \not\equiv 1 \pmod{n}$. Infatti

$$c^{n-1} \equiv a^{n-1}b^{n-1} \equiv a^{n-1} \not\equiv 1 \pmod{n}$$

Inoltre i valori di $c = ab$ al variare di b sono tutti distinti modulo n; infatti se $ab \equiv ab' \pmod{n}$ avremmo che $a(b - b') \equiv 0 \pmod{n}$ da cui $b - b' \equiv 0 \pmod{n}$ in quanto a è relativamente primo con n.

Quindi $a^{n-1} \not\equiv 1 \pmod{n}$ per metà dei possibili valori di a che sono relativamente primi con n. Siccome se a non è relativamente primo con n abbiamo sempre $a^{n-1} \not\equiv 1 \pmod{n}$ possiamo concludere che $a^{n-1} \not\equiv 1 \pmod{n}$ per almeno la metà di tutti i possibili valori di a. □

Dunque se n non è primo e scegliamo a casualmente la probabilità che $a^{n-1} \equiv 1 \pmod{n}$ è minore di $1/2$. Questo ci suggerisce il seguente algoritmo randomizzato per il test di primalità che funziona con una probabilità di errore molto piccola.

PRIMO(n)

1 Scegli un numero costante h di interi casuali $a_1, a_2, \ldots, a_h < n$
2 **if** $a_i^{n-1} \equiv 1 \pmod{n}$ per ogni $i = 1, 2, \ldots, h$
3 **return** SI
4 **else return** NO

La complessità di questo test è $O(hk^3)$, dovuta al calcolo di $a_i^{n-1} \bmod n$ per ogni $i = 1, 2, \ldots, h$.

Se ignoriamo l'esistenza dei numeri di Carmichael l'algoritmo precedente con n primo risponde sicuramente SI mentre, se n è composto risponde NO con probabilità maggiore di $1 - 1/2^h$ e dunque con una probabilità di errore minore di $1/2^h$ che può essere ridotta indefinitivamente aumentando il valore della costante h. Ad esempio con $h = 100$ la probabilità di errore risulta minore di $1/2^{100}$, molto minore della probabilità che durante il calcolo un raggio cosmico colpisca casualmente il computer danneggiandolo.

5 La generazione di numeri primi casuali

La generazione di numeri primi casuali è richiesta in molte applicazioni quali ad esempio lo hashing universale e l'algoritmo randomizzato di Rabin e Karp per il pattern matching su stringhe. La generazione di numeri primi casuali è anche alla base di ogni applicazione crittografica. In questo paragrafo studieremo un efficiente algoritmo randomizzato per generare numeri primi casuali compresi tra 1 ed un intero n generalmente molto grande (per l'algoritmo di Rabin e Karp abbiamo visto che con un processore a 32 bit $n \approx 2^{32}$ ma nelle applicazioni crittografiche possiamo avere anche $n = 2^{4096}$).

Il teorema dei numeri primi ci dice che i primi compresi tra 1 ed n sono $\pi(n) \approx n/\ln(n)$; in altre parole vi è un numero primo ogni $\ln(n)$ interi compresi tra 1 ed n. I numeri primi sono quindi molto abbondanti; ad esempio, poiché $\ln(10000000) \approx 16$, in media uno studente su 14 ha numero di matricola primo.

L'idea è quindi quella di generare un intero p casuale tra 1 ed n e testare se esso è primo; se lo è siamo a posto altrimenti ripetiamo l'operazione con un altro intero casuale. In media dovremo controllare soltanto $\ln(n)$ numeri interi casuali prima di trovarne uno primo.

Quanto veloce è questo metodo per generare numeri primi casuali? Se il numero scelto p è effettivamente primo, il che succede con probabilità $1/\ln n$, esso passa il test. Quindi abbiamo probabilità $1/\ln n$ di trovare un numero primo (e quindi fermarci) per ogni scelta casuale di p. Per il teorema del raccoglitore di figurine occorre testare in media $\ln(n)$ numeri casuali. Dunque il valore atteso del numero di test da effettuare è $\ln(n) = \Theta(k)$ dove k è il numero di cifre binarie di n.

Ma quale tipo di test dobbiamo utilizzare? Siccome p è scelto casualmente non occorre scegliere casualmente i valori di a per randomizzare l'algoritmo; possiamo semplicemente scegliere $a = 2$ (oppure $a = 2, 3, 5$ se vogliamo aumentare la sicurezza). Siccome i numeri sono scelti casualmente il test di Fermat ha una probabilità di errore molto minore di $1/2$. Ad esempio supponiamo di eseguire il test di fermat con $a = 2$ su tutti i numeri $n \leq 25 \times 10^9 \approx 2^{34}$, Tra questi numeri ci sono circa 10^9 primi e soltanto circa 20000 composti che passano il test; quindi la probabilità di dare una risposta errata è circa $20000/10^9 = 2/10^5$.

Questa probabilità di errore diminuisce rapidamente all'aumentare della lunghezza dei numeri per ridursi ad un valore assolutamente trascurabile quando i numeri hanno qualche centinaio di cifre binarie.

Possiamo quindi generare un numero primo casuale di $k > 100$ cifre binarie in tempo medio $O(k^4)$ con una probabilità assolutamente trascurabile che il numero generato non sia primo.

6 Crittografia

Il sistemo di crittografia di Rivest-Shamir-Adelman (RSA) sfrutta la grande differenza tra le complessità polinomiale di certe operazione sui numeri interi (esponenziazione modulare, massimo comun divisore, test di primalità) e la complessità di altre operazioni (fattorizzazione).

Per esporre questo sistema di crittografia consideriamo la situazione in cui due persone Alice e Bruno vogliono comunicare in segreto e una terza persona Carlo cerca di carpire i loro messaggi. Per fissare le idee supponiamo che Alice voglia inviare un messaggio x, scritto in binario, al suo amico Bruno. Essa codifica il messaggio usando una certa funzione di codifica $e(x)$, lo spedisce a Bruno il quale usa la sua funzione di decriptazione $d(.)$ per decodificarlo ottenendo $d(e(x)) = x$. Alice e Bruno sanno che Carlo può intercettare il messaggio codificato $e(x)$. Alice deve scegliere la funzione di codifica $e(.)$ in modo tale che, senza conoscere la funzione di decodifica $d(.)$, Carlo non possa farsene niente del messaggio codificato $e(x)$ da lui intercettato in quanto esso gli da informazione molto scarsa o nulla sul messaggio x.

Per secoli la crittografia si è basata sul cosidetto protocollo a chiave privata. Con questo protocollo Alice e Bruno si sono preventivamente incontrati per concordare un codice con cui criprare i successivi messaggi. In questo caso, l'unica possibilità per Carlo di carpire i messaggi è quella di collezionare un certo numero di messaggi criptati e, analizzandoli, cercare di scoprire il codice.

I metodi a chiave pubblica, quale RSA, sono molto più furbi: permettono ad Alice di inviare un messaggio criptato a Bruno senza che sia necessario averlo incontrato precedentemente per accordarsi sul codice da usare. Questo sembrerebbe a prima vista impossibile: perchè solo Bruno dovrebbe essere in grado di decodificare il messaggio? La soluzione è che Bruno può costruire un lucchetto digitale di cui lui solo possiede la chiave. Rendendo pubblico questo lucchetto digitale egli permette ad Alice di applicare tale lucchetto al messaggio e poi inviarglielo. Carlo, anche se intercetta il messaggio non ha la chiave per aprire il lucchetto, l'unico che può aprire il lucchetto è Bruno che si è tenuta segreta la chiave.

Nel protocollo RSA, Bruno può generare il lucchetto eseguendo delle semplici operazioni. Analogamente sia Alice per codificare un messaggio sia Bruno per decodificarlo devono eseguire operazioni semplici che qualsiasi calcolatore tascabile è in grado di eseguire in pochi istanti. Invece Carlo, per decodificare il messaggio senza possedere la chiave, deve eseguire dei calcoli estremamente complessi che persino sui più potenti supercomputer richiedono molte migliaia di anni per essere eseguiti.

Vediamo ora nel dettaglio come funziona il protocollo RSA. Pensiamo ai messaggi da Alice a Bruno come dei numeri modulo n. Possiamo farlo visto che i messaggi sono

stringhe di cifre binarie di lunghezza minore o uguale ad una certa costante k (se sono più lunghi possiamo sempre spezzarli in messaggi di lunghezza k). Possiamo quindi prendere $n = 2^k$. La funzione di criptazione deve essere una biiezione sull'insieme dei numeri $\{0, 1, \ldots, n-1\}$ altrimenti non esisterebbe la funzione inversa necessaria per la decriptazione. Ma quale valore di n e quale biiezione conviene scegliere? Il seguente risultato ci fornisce una possibile risposta.

Lemma 6.1 *Siano p e q due primi qualsiasi e sia $n = pq$. Per ogni e relativamente primo con $(p-1)(q-1)$ la funzione $e(x) = x^e \bmod n$ è una biiezione sull'insieme $\{0, 1, \ldots, n-1\}$ e la sua inversa è $d(y) = y^d \bmod n$ dove d è l'inverso moltiplicativo di e modulo $(p-1)(q-1)$.*

Dimostrazione. Osserviamo intanto che e, essendo relativamente primo con $(p-1)(q-1)$, ha un inverso moltiplicativo modulo $(p-1)(q-1)$ che può essere calcolato con l'algoritmo esteso di Euclide. Dobbiamo controllare che $d(e(x)) = (x^e)^d = x^{ed} \equiv x \pmod{n}$, ossia che $d(.)$ è effettivamente l'inversa di $e(.)$ (il che implicitamente ci assicura che $e(.)$ è invertibile). Siccome $ed \equiv 1 \pmod{(p-1)(q-1)}$ deve essere $ed = 1 + h(p-1)(q-1)$ per qualche h. Dunque

$$x^{ed} = x^{1+h(p-1)(q-1)} = x^{h(p-1)(q-1)}x$$

Siccome p e q sono primi abbiamo $x^{p-1} \equiv 1 \pmod{p}$ e $x^{q-1} \equiv 1 \pmod{q}$ (per il piccolo teorema di Fermat) e quindi $x^{ed} \equiv x \pmod{p}$ e $x^{ed} \equiv x \pmod{q}$. Dunque $x^{q-1} - x$ è divisibile sia per p che per q ed, essendo p e q primi, è divisibile anche per $n = pq$. Pertanto $d(e(x)) \equiv x \pmod{n}$. □

Il lemma precedente dice che la funzione $e(x) = x^e \pmod{n}$ è biunivoca e si può calcolare in tempo $O(\log^3 n)$. Essa è quindi un modo ragionevole di codificare il messaggio x — non vengono perse informazioni.

Bruno può quindi generare i due numeri primi p e q rappresentabili con k cifre binarie (tempo $O(k^4)$), scegliere e relativamente primo con $(p-1)(q-1)$, ad esempio $e = 3$, e calcolare il suo inverso d modulo $(p-1)(q-1)$ (tempo $O(k^3)$), calcolare $n = pq$ (tempo $O(k^2)$) ed infine pubblicare la coppia (n, e) mantenendo segreto d.

Alice, per spedire un messaggio x a Bruno deve soltanto calcolare $y = e(x) = x^e \bmod n$ (tempo $O(k^3)$).

Bruno, conoscendo d, può decodificare il messaggio codificato y semplicemente calcolando $x = d(y) = y^d \bmod n$ (tempo $O(k^3)$).

Carlo invece, non conoscendo d, non può decodificare y. Per decodificarlo dovrebbe prima calcolare d conoscendo e ed n e per fare questo deve scomporre n per determinare i due numeri primi p e q necessari per calcolare l'inverso moltiplicativo di d. Ma n è un numero di $2k$ cifre binarie e calcolare la sua scomposizione in fattori primi richiede tempo $O(\sqrt{n}) = O(2^k)$, assolutamente impossibile se n ha almeno qualche centinaio di cifre binarie.

Note

L'algoritmo randomizzato per il test di primalità che abbiamo descritto è stato ideato da Miller [] e Rabin [] ed è attualmente l'algoritmo più veloce, a meno di fattori costanti.

Per molti anni il test di primalità è stato un esempio in cui la randomizzazione sembrava necessaria per ottenere un algoritmo efficiente (ossia polinomiale nel numero k di cifre binarie di n). Nel 2002 Agrawal, Kayal e Saxema [] hanno sorpreso tutti con il loro algoritmo deterministico polinomiale.

Fino a quel momento il miglior algoritmo deterministico era quello dovuto a Coen e Lestra [] che richiedeva tempo $O(k^{\log \log k})$, appena un poco superpolinomiale.

Nondimeno, ai fini pratici gli algoritmi randomizzati risultano più veloci e quindi da preferire.

Il problema della generazione casuale di numeri primi è ben descritto in un articolo di Beauchemin, Brassard, Crépeau, Goutier e Pomerance [].

Il test di primalità si esegue in tempo polinomiale ma, nel caso in cui il numero non sia primo non ci fornisce la sua scomposizione in fattori.

Tutti gli algoritmi noti per fattorizzare un intero n grande hanno un tempo di esecuzione $\Omega(2^{\sqrt[3]{k}})$ che cresce esponenzialmente con la radice cubica del numero k di cifre binarie di n.