6 Complessità computazionale di un algoritmo

6.1 Rappresentazione di un intero in base b

Definizione 6.1. **(Rappresentazione di n in base b)**
Siano n, b interi non negativi, allora $n = d_{k-1}b^{k-1} + ... + d_1 b + d_0$, dove $0 \leq d_{k-1}, ...d_1, d_0 \leq b-1$. La notazione $(d_{k-1}...d_1 d_0)_b$ indica la **rappresentazione di n in base b** e le gli interi $d_{k-1}, ...d_1, d_0$ sono le **cifre di n in base b**.

Se $b = 2$, allora le cifre in binario si dicono **bit** (binary digit).

Esempio 6.2. Esprimiamo $n = 1000$ in base 2, 7 e 26.

1. $b = 2$, allora $n = 2^9 + 2^8 + 2^7 + 2^6 + 2^5 + 2^3$ e quindi $n = 1111101000$.
2. $b = 7$, allora $n = 2 * 7^3 + 6 * 7^2 + 2 * 7 + 6$ e $n = (2626)_7$.
3. $b = 26$, usando le lettere dell'alfabeto inglese come cifre $A = 0, \cdots, Z = 25$, si ha $n = (BLC)_{26}$.

Si noti che se $n = (d_{k-1}...d_1 d_0)_b$, vale che $b^{k-1} \leq n < b^k$ ed il numero delle cifre di n in base b è

$$k = \lfloor \log_b n \rfloor + 1.$$

Esempio 6.3. Dall'esempio precedente segue che $n = 1000$ in base 2, 7 e 26 ha il seguente numero di cifre:

1. $n = 1111101000$ e $\lfloor \log_2 10^3 \rfloor + 1 = 10$.
2. $n = (2626)_7$ e $\lfloor \log_7 10^3 \rfloor + 1 = 4$.
3. $n = (BLC)_{26}$ e $\lfloor \log_{26} 10^3 \rfloor + 1 = 3$.

6.2 Complessità temporale di un algoritmo

L'efficienza di un algoritmo nella risoluzione di un problema si misura in base a

1. Il **tempo** richiesto per eseguire le operazioni elementari.
2. Lo **spazio** necessario per memorizzare e manipolare dati.

Il tempo è la risorsa più importante deve essere indipendente dalle performance del calcolatore utilizzato. Un modo oggettivo per la misura del tempo di calcolo di un algoritmo è quello di esprimerlo in funzione della dimensione n dei dati in input.

Definizione 6.4. **(Complessità computazionale di un algoritmo)** La **complessità computazionale (temporale)** di un algoritmo è una funzione $T(n)$ dove n è la dimensione dei dati di ingesso.

Siccome vale che $\lim_{n\to+\infty} T(n) = +\infty$, per stimare T è importante l' ordine di infinito di T. Per fare ciò si consideri la seguente definizione.

Definizione 6.5. **(Notazione O-grande)** Siano $f, g : \mathbb{N}^r \longrightarrow \mathbb{R}$, allora $f = O(g)$ se, e solo se, esistono due costanti $B, C \in \mathbb{R}$ tali che per ogni $j = 1, ...r$, se $n_j > B$ allora

$$f(n_1, ..., n_r) < Cg(n_1, ..., n_r).$$

Esempio 6.6. Valgono i seguenti fatti:

1. $(r = 1)$, se $f(n) = 2n^2 + 3n - 3$, allora $f(n) < 3n^2$ e quindi $f = O(n^2)$.
2. Siano $f, g : \mathbb{N} \longrightarrow \mathbb{R}$, se $\lim_{n\to+\infty} \frac{f(n)}{g(n)} = \ell$ e $\ell \in \mathbb{R}$, allora $f = O(g)$. Infatti, fissato $\varepsilon = \ell/2$ per esempio, esiste $B > 0$ tale che per ogni $n > B$ risulta $\left|\frac{f(n)}{g(n)} - \ell\right| < \frac{\ell}{2}$ e quindi $f(n) < \frac{3}{2}\ell g(n)$.
3. Per ogni $\varepsilon > 0$ vale che $\ln n = O(n^\varepsilon)$. Segue da (1) tenendo presente che $\lim_{n\to+\infty} \frac{\ln n}{n^\epsilon} = 0$, per esempio, per la Teorema di de l'Hôpital.
4. Sia $f(n, b) = \lfloor \log_b n \rfloor + 1$, allora $f(n, b) < 2 \log_b n$ e quindi $f(b, n) = O(\log_b n)$.

Definizione 6.7. **(Complessità computazionale polinomiale di un algoritmo)**
Un algoritmo che ha come input gli interi $n_1, n_2..., n_r$ di $k_1, k_2..., k_r$ bit, rispettivamente, si dice avere **complessità computazionale polinomiale**, se esistono degli interi $d_1, d_2..., d_r$ tali che

$$T(n) = O(k_1^{d_1} k_2^{d_2} \cdots k_r^{d_r}).$$

6.3 Bit Operazioni

In questo sezione intendiamo stimare il tempo di esecuzione di semplici algoritmi utilizzati nell'ambito della teoria dei numeri.

6.3.1 Somma di due numeri di lunghezza binaria k.

Per esempio, si opera come segue.

In realtà dobbiamo ripetere questo procedimento k volte:

1. Se entrambi i bit sono 0 e non ci sta riporto, si segna 0 e si prosegue.
2. Se entrambi i bit sono 0 e ci sta riporto di 1, oppure uno dei due bit è 0, altro è 1 e non ci sta riporto, allora si scrive 1 e si prosegue
3. Se uno dei due bit è 0, altro è 1 e ci sta il riporto di 1, oppure entrambi sono 1 e non ci sta riporto, allora si scrive 0 e si mette il riporto di 1 alla colonna successiva.
4. Se entrambi i bit sono 1 e ci sta il riporto di 1, si scrive 1 e mette il riporto di 1 alla colonna successiva.

$$\begin{array}{r} {\scriptstyle 1\ 1\ 1\ 1\ } \\ 1111000 \\ +\ \underline{0011110} \\ 10010110 \end{array}$$

Ognuno di questi quattro passaggi è chiamato **operazione bit**. Quindi, sommare due numeri, entrambi di lunghezza binaria k, richiede k operazioni.

6.3.2 Prodotto di due numeri n e m di lunghezza binaria k e ℓ, rispettivamente, con $k \geq \ell$.

Molte delle stime per il calcolo della complessità computazionale di operazioni elementari fatte qui di seguito non sono ottimali, ma hanno tuttavia un valore didascalico. Per esempio, si moltiplichi 11101 per 1101.

$$\begin{array}{r} 11101 \\ \underline{\quad 1101} \\ 11101 \\ 111010 \\ \underline{11101\quad\ } \\ 101111001 \end{array}$$

Dalla figura si evince che, se $\ell' \leq \ell$ è il numero delle cifre non nulle di m, allora si riportano ℓ' copie di n, ognuna shiftata verso sinistra di una posizione rispetto alla precedente.

Siccome vogliamo valutare la moltiplicazione di n per m, non possiamo sommare le ℓ' copie shifate di n ma dobbiamo procedere in modo induttivo: sommiamo le prime due, poi sommiamo ogni riga alla somma parziale delle precedenti. Quindi eseguiamo $\ell' - 1$ addizioni.

Siccome ognuna di queste prevede k operazioni bit, allora

$$T(n \times m) < k\ell < k^2.$$

Esercizio 6.8. Provare che $T(\lfloor \sqrt{n} \rfloor) = O(\log_2 n)^3$.

Dimostrazione. Supponiamo che n abbia $k+1$ bit. Se $m = \lfloor \sqrt{n} \rfloor$, allora $m = 2^{\lfloor k/2 \rfloor} + \sum_{i=0}^{\lfloor k/2 \rfloor - 1} x_i 2^i$.
Allora si opera come segue:

1. Per ogni $j = 1, ..., \lfloor k/2 \rfloor$ si testa $m^2 \leq n$ per m ottenuto in corrispondenza delle $(\lfloor k/2 \rfloor + 1)$-upla $\left(x_{\lfloor k/2 \rfloor}, x_{\lfloor k/2 \rfloor - 1}, ..., x_0\right)$ tale che $x_{\lfloor k/2 \rfloor - e} = 1$ per $e \leq j$ e $x_{\lfloor k/2 \rfloor - e} = 0$ per $e \geq j + 1$.
2. Quindi si eseguono al più $\lfloor k/2 \rfloor + 1$ moltiplicazioni di numero con sè stesso, avente al più $\lfloor k/2 \rfloor$ cifre.
3. Il numero delle bit operazioni eseguite è al più $(\lfloor k/2 \rfloor + 1)(\lfloor k/2 \rfloor)^2 \leq k^3$.

Pertanto $T(\lfloor \sqrt{n} \rfloor) = O(\log_2 n)^3$.

□

Esercizio 6.9. Provare che $T(n!) = O((n-2)n\left(\lfloor \log_2 n \rfloor + 1\right)^2)$.

Dimostrazione. Per calcolare $n! = 2 \cdot 3 \cdots (n-1) \cdot n$ si procede come segue. Si calcola $2 \cdot 3$, il risultato lo si moltiplica per 4, il risultato per 5. Essendo i numeri $n-1$, allora il numero delle moltiplicazioni che si esegue è $n-2$.

1. Per $j = 1, \ldots, n-1$, al passo $(j-1)$-esimo, si moltiplica $j!$ per $j+1$. Se $k = \lfloor \log_2 n \rfloor + 1$, per ogni $2 \leq j \leq n$ risulta $\lfloor \log_2 j \rfloor + 1 \leq k$ e $\lfloor \log_2 (j!) \rfloor + 1 \leq nk$.

2. Il numero delle bit operazioni necessarie per calcolare $j!$ per $j+1$ è minore di $nk \cdot k = nk^2$.

3. Siccome si eseguono $n-2$ moltiplicazioni nel punto **(2)**, allora
$$T(n!) = O((n-2)nk^2).$$

4. Per $n \to +\infty$ vale che $(n-2)n\left(\lfloor \log_2 n \rfloor + 1\right)^2 \simeq n^2 \left(\log_2 n\right)^2$.

□

Esercizio 6.10. Provare che $T(\binom{n}{m}) = O(m^2 \left(\log_2 n\right)^2)$.

Dimostrazione. Siccome $\binom{n}{m} = \binom{n}{n-m}$ possiamo assumere $m \leq n/2$. Inoltre, poiché
$$\binom{n}{m} = \frac{D_{n,m}}{2 \cdot 3 \cdots (m-1) \cdot m},$$
dove $D_{n,m} = n \cdot (n-1) \cdots (n-m+1)$, per determinare $\binom{n}{m}$ è necessario eseguire $m-1$ moltiplicazioni e successivamente $m-1$ divisioni.

1. Per $j = 1, \ldots, m-1$, al passo $(j-1)$-esimo si moltiplica $D_{n,j}$ per $n-j$, dove $D_{n,j} \leq D_{n,m} \leq n^m$.

2. Se $k = \lfloor \log_2 n \rfloor + 1$, allora
$$\lfloor \log_2 D_{n,j} \rfloor + 1 \leq \lfloor \log_2 D_{n,m} \rfloor + 1 \leq \lfloor \log_2 n^m \rfloor + 1 = mk.$$

3. Per $j = 1, \ldots, m-1$, al passo $(j-1)$-esimo si eseguono al più mk^2 bit operazioni. Pertanto il numero delle bit operazioni per determinare $D_{n,m}$ è al più $(m-1)mk^2$.

4. Determinato $D_{n,m}$, si eseguono $m-1$ divisioni tra $D_{n,m}$ e i $2 \leq e \leq m \leq n/2$, ognuna delle quali impiega al più $(mk)\,k = mk^2$ bit operazioni.

   Quindi il numero delle bit operazioni necessario per dividere $D_{n,m}$ con $m!$ è $(m-1)mk^2$.

5. $T(\binom{n}{m}) < (m-1)mk^2 + (m-1)mk^2 < 2m^2k^2$ e $2m^2k^2 \simeq 2m^2 \left(\log_2 n\right)^2$ per $n \to +\infty$, che è l'asserto.

□

Esercizio 6.11. Provare che $T(n_b) = O((\ln n)^2)$.

Dimostrazione. Per convertire un intero scritto in binario n e avente k bit in base b bisogna dividere per $b = (b_{\ell-1}...b_0)_2$. Il resto d_0 sarà uno degli interi da 0 a $b-1$. Quindi $n = n_1 b + d_0$. Ora rimpiazziamo n con n_1 e ripetiamo l'operazione.

1. Il numero delle divisioni da eseguire è uguale al numero delle cifre in base b di n che è
$$\left\lfloor \frac{\ln n}{\ln b} \right\rfloor + 1 = \left\lfloor \frac{\ln n}{\ln 2} \cdot \left(\frac{\ln b}{\ln 2} \right)^{-1} \right\rfloor + 1 = O(k/\ell).$$
2. Il numero dei bit operazioni impiegati per eseguire la divisione di numeri (i quozienti) minori o uguali a n con b è $O(k\ell)$.
3. $T(n_b) = O(k/\ell) \cdot O(k\ell) = O(k^2) = O((\ln n)^2)$.

□

Si noti che il tempo non dipende dalla grandezza della base b scelta. Ci ò avviene perché il maggior tempo utilizzato per determinare ogni cifra in base b è compensato dal minor numero di cifre da determinare.

Quindi il numero delle bit operazioni impiegate per eseguire la divisione è k^2. Il numero dei passai da eseguire è uguale al numero delle cifre decimale di n che è $\left\lfloor \frac{\ln n}{\ln 10} \right\rfloor + 1 = O(k)$, essendo $k = \left\lfloor \frac{\ln n}{\ln 2} \right\rfloor + 1$ avente k bit.

Esercizio 6.12. Siano $P(X) = \sum_{i=0}^{n_1} a_i X^i$ e $Q(X) = \sum_{j=0}^{n_2} b_j X^j$, $n_2 \le n_1$, tali che $a_i, b_j \in \mathbb{R}$ e $a_i, b_j \le m$, allora $T(P(X)Q(X)) = O(n^2 (\ln m)^2)$, dove n denota n_1.

Dimostrazione. Sia $R(X) = P(X)Q(X)$, allora $R(X) = \sum_{v=0}^{n_1+n_2} c_v X^v$, dove $c_v = \sum_{i+j=v} a_i b_j$. Il numero degli addendi in c_v è al più uguale al numero dei b_j che è $n_2 + 1$. Quindi per determinare c_v occorrono al più n_2 addizioni e $n_2 + 1$ moltiplicazioni.

1. Per eseguire le n_2 moltiplicazioni occorrono $(n_2 + 1)(\lfloor \log_2 m \rfloor + 1)^2$ bit operazioni.
2. Poiché $a_i b_j \le m^2$ allora $c_v \le n_2 m^2$ e quindi ogni somma parziale è minore o uguale di $n_2 m^2$. Quindi il numero delle bit operazioni necessario per eseguire una addizione è minore o uguale al numero delle bit operazioni necessario per eseguire $n_2 m^2 + m^2$, che è $(\lfloor \log_2 n_2 m^2 \rfloor + 1)$. Quindi il per eseguire le n_2 addizioni sono necessarie al più $n_2(\lfloor \log_2 n_2 m^2 \rfloor + 1)$.
3. Il numero totale delle bit operazioni per determinare c_v è al più
$$(n_2 + 1)(\lfloor \log_2 m \rfloor + 1)^2 + n_2(\lfloor \log_2 n_2 m^2 \rfloor + 1).$$
4. il numero dei c_v è al più $n_1 + n_2 + 1$, quindi il numero N di bit operazioni utilizzati per determinare $R(X)$ è
$$N \le (n_1 + n_2 + 1)\left[(n_2 + 1)(\lfloor \log_2 m \rfloor + 1)^2 + n_2(\lfloor \log_2 n_2 m^2 \rfloor + 1)\right].$$

5. Siccome $n_2 \leq n_1$, posto $n = n_1$, per $m \geq 16$ and $m \geq \sqrt{n_2}$ che si verificano nella realtà essendo $m \to +\infty$

$$N \leq (2n+1)\left[(n+1)(\log_2 m + 1)^2 + n(3\log_2 m + 1)\right]$$
$$\simeq 2n^2 (\log_2 m)^2 = \frac{2}{\ln 2} n^2 (\ln m)^2$$

per $n, m \to +\infty$.

Pertanto $T(P(X)Q(X)) = O(n^2 (\ln m)^2)$.

□

6.3.3 Algoritmo Euclideo

L'**Algoritmo Euclideo** è usato per il calcolo del massimo comune divisore tra due interi a e b.

Per semplicità assumiamo $0 < b \leq a$. Posto $r_0 = a$ e $r_1 = b$, esso opera come segue:

$$\begin{array}{ll} r_0 = q_1 r_1 + r_2 & 0 < r_2 < r_1 \\ r_1 = q_2 r_2 + r_3 & 0 < r_3 < r_2 \\ \vdots & \vdots \\ r_{m-2} = q_{m-1} r_{m-1} + r_m & 0 < r_m < r_{m-1} \\ r_{m-1} = q_m r_m & \end{array}$$

Lo pseudocodice dell'Algoritmo Euclideo è

Algoritmo 6.13. **(Algoritmo Euclideo (a, b))**

$r_0 \longleftarrow a$
$r_1 \longleftarrow b$
$m \longleftarrow 1$
while $r_m \neq 0$
do $\begin{cases} q_m \longleftarrow \left\lfloor \frac{r_{m-1}}{r_m} \right\rfloor \\ r_{m+1} \longleftarrow r_{m-1} - q_m r_m \\ m \longleftarrow m + 1 \end{cases}$
$m \longleftarrow m - 1$
return $(q_1, ..., q_m; r_m)$
comment $r_m = \gcd(a, b)$

Proposizione 6.14. Valgono i seguenti fatti:

(*i*) $\gcd(a,b) = r_m$.
(*ii*) $m \leq 2\lfloor \log_2 a \rfloor + 1$.
(*iii*) Se $a > b$, $T(\gcd(a,b)) = O(\ln^3 a)$.

Dimostrazione.

(*i*) Se t è un intero, poiché $r_i = q_{i+1}r_{i+1} + r_{i+2}$ allora risulta

$$t \mid \gcd(r_i, r_{i+1}) \iff t \mid r_i \text{ e } t \mid r_{i+1} \iff t \mid r_{i+1} \text{ e } t \mid r_{i+2}$$
$$\iff t \mid \gcd(r_{i+1}, r_{i+2}).$$

Pertanto

$$\gcd(a,b) = \gcd(r_0, r_1) = \gcd(r_1, r_2) = \cdots = \gcd(r_{m-1}, r_m) = r_m.$$

(*ii*) Se $r_{i+1} \leq \frac{1}{2}r_i$, allora $r_{i+2} < r_{i+1} \leq \frac{1}{2}r_i$. Se, invece $r_{i+1} > \frac{1}{2}r_i$, allora si ha $r_i = r_{i+1} + r_{i+2}$ (i.e. $q_{i+1} = 1$) e quindi, $r_{i+2} = r_i - r_{i+1} < r_i - \frac{1}{2}r_i = \frac{1}{2}r_i$.

Quindi, per ogni $i = 0, \ldots m-2$ vale che $r_{i+2} < \frac{1}{2}r_i$. Pertanto, $1 \leq r_m < \frac{1}{2^{\lfloor m/2 \rfloor}} r_0$ e quindi $\lfloor m/2 \rfloor < \log_2 a$. Ciò implica $m \leq 2\lfloor \log_2 a \rfloor + 1$

(*iii*) Poichè $r_i \leq a$, ogni divisione tra r_i e r_{i+1} impiega $O(\ln^2 a)$ bit operazioni, e poichè il numero delle divisioni da effettuare è $m \leq 2\lfloor \log_2 a \rfloor + 1 = O(\ln a)$, allora il tempo impiegato dall'**Algoritmo 6.13** per determinare $\gcd(a,b)$ è $O(\ln^2 a)O(\ln a) = O(\ln^3 a)$.

□

Remark 6.15. In realtà si prova che, se $a > b$, il tempo impiegato dall'**Algoritmo 6.13** per determinare $\gcd(a,b)$ è $O(\ln^2 a)$.

Siano a e b interi tali che $0 < b \leq a$, e siano $r_1, \ldots, r_m$ e $q_1, \ldots, q_m$ gli interi generati dall'algoritmo euclideo. Si definiscano gli interi $t_1, \ldots, t_m$ e $s_1, \ldots, s_m$ come segue:

$$t_j = \begin{cases} 0 & \text{se } j = 0 \\ 1 & \text{se } j = 1 \\ t_{j-2} - q_{j-1}t_{j-1} & \text{se } j \geq 2 \end{cases}$$

$$s_j = \begin{cases} 1 & \text{se } j = 0 \\ 0 & \text{se } j = 1 \\ s_{j-2} - q_{j-1}s_{j-1} & \text{se } j \geq 2 \end{cases}$$

Allora vale la seguente

Proposizione 6.16. Per ogni $j = 0, ..., m$ risulta $r_j = s_j r_0 + t_j r_1$.

Dimostrazione. Procediamo per induzione su j. La tesi è banalmente vera per $j = 0$ e per $j = 1$. Ora, si supponga vera la tesi per $j \leq i - 1$ e la si dimostri per $j = i$. Siccome, $r_{i-2} = q_{i-1} r_{i-1} + r_i$, sfruttando l'ipotesi induttiva si ha:

$$\begin{aligned} r_i &= r_{i-2} - q_{i-1} r_{i-1} = (s_{i-2} r_0 + t_{i-2} r_1) - q_{i-1} (s_{i-1} r_0 + t_{i-1} r_1) \\ &= (s_{i-2} - q_{i-1} s_{i-1}) r_0 + (t_{i-2} - q_{i-1} t_{i-1}) r_1 = s_i r_0 + t_i r_1, \end{aligned}$$

che è l'asserto.

□

Algoritmo 6.17. **(Algoritmo Euclideo Esteso (a, b))**

$a_0 \longleftarrow a$
$b_0 \longleftarrow b$
$t_0 \longleftarrow 0$
$t \longleftarrow 1$
$s_0 \longleftarrow 1$
$s \longleftarrow 0$
$q \longleftarrow \left\lfloor \frac{a_0}{b_0} \right\rfloor$
$r \longleftarrow a_0 - q b_0$
while $r > 0$
do
$temp \longleftarrow t_0 - q t_0$
$t_0 \longleftarrow t$
$t \longleftarrow temp$
$s_0 \longleftarrow s$
$s \longleftarrow temp$
$a_0 \longleftarrow b_0$
$b_0 \longleftarrow r$
$q \longleftarrow \left\lfloor \frac{a_0}{b_0} \right\rfloor$
$r \longleftarrow a_0 - q b_0$
return (r, s, t)
comment $r = \gcd(a, b)$ and $r = sa + tb$

Corollario 6.18. *Se* $\gcd(r_0, r_1) = 1$*, allora* t_m *è l'inverso di* r_1 *modulo* r_0*.*

Dimostrazione. Dalla Proposizione precedente, vale che

$$s_m r_0 + t_m r_1 = r_m = 1,$$

essendo $r_m = \gcd(r_0, r_1)$ come conseguenza dell'algoritmo euclideo.
Quindi $t_m r_1 \equiv 1 \ mod\ r_0$, ovvero la tesi.

□

Algoritmo 6.19. **(Inverso moltiplicativo** (a, b)**)**

$a_0 \longleftarrow a$
$b_0 \longleftarrow b$
$t_0 \longleftarrow 0$
$t \longleftarrow 1$
$q \longleftarrow \left\lfloor \frac{a_0}{b_0} \right\rfloor$
$r \longleftarrow a_0 - qb_0$
while $r > 0$
do
- $temp \longleftarrow (t_0 - qt_0)\ mod\ a$
- $t_0 \longleftarrow t$
- $t \longleftarrow temp$
- $a_0 \longleftarrow b_0$
- $b_0 \longleftarrow r$
- $q \longleftarrow \left\lfloor \frac{a_0}{b_0} \right\rfloor$
- $r \longleftarrow a_0 - qb_0$

if $r \neq 1$
then b has no inverse modulo n
else return (t)

Corollario 6.20. *Se* $\gcd(r_0, r_1) = 1$*, allora l'inverso di* r_1 *modulo* r_0 *è determinato in* $O(\ln^3 a)$ *operazioni bit.*

Dimostrazione. Per il calcolo dei r_j,q_j,t_j e s_j vengono eseguite $O(\ln^2 a)$ operazioni bit, $O(\ln a)$ volte. Pertanto, per il calcolo t_m vengono eseguite $O(\ln^3 a)$ operazioni bit.

□

Esempio 6.21. Vogliamo determinare l'inverso di 28 modulo 75.
Posto $r_0 = 75$ e $r_1 = 28$, vale che $75 = 2 * 28 + 19$. Quindi $q_1 = 2$ e r_2

i	r_i	q_i	s_i	t_i
0	75	–	1	0
1	28	2	0	1
2	19	1	1	-2
3	9	2	-1	3
4	1	9	3	-8

Pertanto $3 * 75 + (-8) * 28 = 1$ e quindi $(-8) * 28 \equiv 1 \ mod\ 75$. Siccome $-8 \equiv 67 \ mod\ 75$, vale che 67 è l'inverso di 28 mod 75.

□

6.3.4 Complessità delle operazioni in $\mathbb{Z}_n$

In questo paragrafo forniamo il tempo impiegato per compiere le operazioni in $\mathbb{Z}_n$.

Sia k il numero delle cifre in binario di n e siano a, b due interi tali che $0 \leq a, b \leq n-1$.

- $T((a \pm b) \ \textbf{mod}\ n) = O(k)$**.**

  Proviamo solo che Tempo$((a \pm b) mod\ n) = O(k)$ (la differenza per esercizio). Siccome $0 \leq a, b \leq n-1$, allora $0 \leq a+b \leq 2n-2$. Quindi se $a+b \leq n-1$, allora $(a+b) mod\ n = a+b$, se $a+b \geq n$, allora $(a+b) mod\ n = a+b-n$. Quindi in ogni caso vengono impiegate $O(k)$ bit operazioni.

- $T(ab \ \textbf{mod}\ n) = O(k^2)$.

  Per il calcolo di ab vengono impiegate al più k^2 bit operazioni. Inoltre ab è un numero di $2k$ bit quindi la successiva divisione per k impiega $4k^2$ bit. Quindi il calcolo di $ab \ mod\ n$ impiega $O(k^2)$ bit.

- **Se** $\gcd(a, n) = 1$**, allora** $T(a^{-1} \ \textbf{mod}\ n) = O(k^3)$**.**

  Vero per il **Corollario 6.20**.

- $T(a^b \ \textbf{mod}\ n) = O(k^2 \log b)$**.**

  In particolare, se $b < n$**, allora** $T(a^b \ \textbf{mod}\ n) = O(k^3)$**.**

  Sia $b = \sum_{i=0}^{\ell-1} b_i 2^i$ con $b_i = 0, 1$, allora il calcolo di $a^b \ mod\ n$ avviene attraverso il Metodo dei Quadrati Ripetuti.

Algoritmo 6.22. **(Quadrati Ripetuti** (a, b, n)**)**

$z \longleftarrow 1$
for $i \longleftarrow \ell - 1$ **downto** 0
do { $z \longleftarrow z^2 \ mod \ n$
if $b_i = 1$
then $z \longleftarrow (z \times a) \ mod \ n$
return (z)

Siccome ad ogni passo si valuta $z^2 \ mod \ n$ e poi, eventualmente, $(z \times a) \ mod \ n$ vengono utilizzati $O(k^2)$ bit poichè $z^2 \ mod \ n$ è costituito da al più k bit.
I due prodotti $z^2 \ mod \ n$ e $(z \times a) \ mod \ n$ sono ripetuti al più ℓ volte. Pertanto, $T(a^b \ mod \ n) = O(k^2 \log b)$. In particolare, se $b < n$, allora $T(a^b \ mod \ n) = O(k^3)$.

Esempio 6.23. Calcolare $a^b \ mod \ n$, dove $a = 9726$, $b = 3533$ e $n = 11413$.

Siccome $b = 2^0 + 2^2 + 2^3 + 2^6 + 2^7 + 2^8 + 2^{10} + 2^{11}$, allora

i	b_i	$z \ mod \ n$
11	1	$1^2 \times 9726 = 9726$
10	1	$9726^2 \times 9726 = 2559$
9	0	$2559^2 = 5634$
8	1	$5634^2 \times 9726 = 9167$
7	1	$9167^2 \times 9726 = 4958$
6	1	$4958^2 \times 9726 = 7783$
5	0	$7783^2 = 6298$
4	0	$6298^2 = 4629$
3	1	$4629^2 \times 9726 = 10185$
2	1	$10185^2 \times 9726 = 105$
1	0	$105^2 = 11025$
0	1	$11025^2 \times 9726 = 5761$.

Pertanto, $9726^{3533} \ mod \ 11413 = 5761$.

□