Convenzione: Su questo foglio $\log n$ indica sempre il logaritmo in base 2 di n.

1. Quante cifre binarie ha all'incirca un numero intero n ?

Sol. Scriviamo n in forma binaria: $(a_{k-1}a_{k-2}\dots a_1a_0)_2$, con $a_i \in \{0,1\}$ e $a_{k-1} \neq 0$. Poiché $n = a_{k-1}2^{k-1} + a_{k-2}2^{k-2} + \dots + a_1 2 + a_0$, abbiamo

$$2^{k-1} \leq n < 2^k.$$

Facendo il logaritmo in base 2 di tutti i termini troviamo $k-1 \leq \log n \leq k$, da cui segue che il numero k di cifre binarie di n soddisfa la relazione

$$\log n \leq k \leq 1 + \log n.$$

Dunque k vale all'incirca $\log n$: precisamente $k = [\log n] + 1$.

2. Quante cifre decimali ha un numero n di $\log n$ cifre binarie?

Sol. Con lo stesso ragionamento troviamo che il numero di cifre decimali di n è all'incirca

$$\log_{10} n = \frac{\log n}{\log 10} \sim \frac{1}{3}\log n, \qquad \log 10 = 3,322\dots$$

cioè circa un terzo del numero di cifre binarie di n.

3. Fra i numeri interi compresi fra 0 e 1000000, quanti hanno almeno 6 cifre?

Sol. Un numero compreso fra 0 e 1000000, di almeno sei cifre decimali, è un numero $100000 \leq n \leq 1000000$. Ce ne sono $1000000 - 99999 = 900001$, quindi sono il 90% dei umero fra 0 e 1000000.

4. Siano n ed m due numeri interi rispettivamente di k ed h cifre binarie. Quante cifre binarie hanno $n+m$, $-n$, $n-m$, $n \cdot m$, n^a, con $a \in \mathbb{N}$?

Sol. Supponiamo ad esempio che $n \geq m$. Allora $n+m$ ha al più $k+1$ cifre binarie; $-n$ è il complemento a due di n (si trova sommando 1 al complemento a 1 di n, ossia la stringa con zeri al posto degli uni e uni al posto degli zeri), dunque ha lo stesso numero di cifre binarie di n; per quanto detto al punto precedente, è chiaro che $n-m$ ha al più k cifre binarie; $n \cdot m$ è ottenuto sommando m righe di lunghezza n, ognuna traslata di un bit verso sinistra rispetto alla precedente.

$$\begin{array}{r}
a_{k-1}a_{k-2}\dots\dots a_1a_0\times \\
b_{h-1}b_{h-2}\dots b_1b_0 = \\
- - - - - - - - - - - - - \\
l_{k-1}l_{k-2}\dots\dots l_1l_0 \\
m_{k-1}m_{k-2}\dots\dots m_1m_0- \\
\dots\ \dots\ \dots\ \dots\ \dots\ \dots - \ -
\end{array}$$

Dunque ha circa $k+h$ cifre binarie; da ciò segue che n^a ha circa $a \cdot k$ cifre binarie.

5. Sia n un intero di 100 cifre decimali. Quante cifre hanno all'incirca $\sqrt{n}$, n^3, $n + \sqrt{n}$? Di quante cifre differiscono all'incirca n, $n + \sqrt{n}$ e $n - \sqrt{n}$?

Sol. $\sqrt{n}$ ha all'incirca 50 cifre decimali, n^3 ne ha circa 300, $n + \sqrt{n}$ ne ha circa 100. Quindi n, $n + \sqrt{n}$ e $n - \sqrt{n}$ sono numeri di circa 100 cifre che grossomodo differiscono fra loro per le ultime 50 cifre.

Complessità

Un algoritmo è una sequenza di passi che a partire un *input* produce un *output*. La *complessità di un algoritmo* è il numero di operazioni elementari (sui singoli bit) da esso richieste per produrre l'output, in funzione dell'ordine di grandezza dell'input. è importante determinare il comportamento asintotico di questa funzione, cioè al crescere dell'ordine di grandezza dell'input.

Negli algoritmi che consideremo in seguito l'input è costituito da numeri interi. L'ordine di grandezza di un intero n è misurato dal numero di cifre, che è circa $\log_{10} n$, o dal numero di bits che è circa $\log_2 n$. In questo contesto:

Definizione Un algoritmo si dice *polinomiale* se la sua complessità è una funzione polinomiale nel numero di cifre degli elementi dell'input; si dice *esponenziale* se la sua complessità è una funzione esponenziale nel numero di cifre di almeno uno degli elementi dell'input; si dice *subesponenziale* se la sua complessità è una funzione subesponenziale nel numero di cifre di alcuni elementi dell'input, e polinomiale nel numero di cifre dei rimanenti.

Esempio. Supponiamo che un dato algoritmo abbia come input un numero intero n e indichiamo con $\phi(n)$ la sua complessità.
- La sua complessità è polinomiale se esistono costanti C ed N tali che, per ogni $n > N$,

$$|\phi(n)| \leq C|\log^k n|, \qquad k \in \mathbb{N}_{\geq 1}.$$

In tal caso diciamo che è $\mathcal{O}(\log^k n)$.
- La sua complessità è esponenziale se esistono costanti C ed N tali che, per ogni $n > N$,

$$|\phi(n)| \leq C|2^{k \log n}|, \qquad k \in \mathbb{N}_{\geq 1}.$$

In tal caso diciamo che è $\mathcal{O}(2^{k \log n})$.
- La sua complessità è subesponenziale se $\phi(n)$ è $\mathcal{O}(2^{(\log n)^\alpha})$, per $0 < \alpha < 1$. Ad esempio $\phi(n) = 2^{\sqrt{\log n}}$ è subesponenziale.

La grossa differenza fra complessità polinomiale ed esponenziale (o subesponenziale) è la seguente:

supponiamo che il numero di operazioni richieste dall'algoritmo per un numero di $\log n$ cifre sia polinomiale dell'ordine di $\log^k n$. Allora il numero di operazioni richieste dall'algoritmo per un numero di $2 \log n$ cifre è dell'ordine di $2^k \log^k n$. In altre parole, raddoppiando il numero di cifre dell'input, il numero di operazioni richieste si moltiplica per un fattore 2^k che *non dipende dalla grandezza dell'input.*

supponiamo che il numero di operazioni richieste dall'algoritmo per un numero di $\log n$ cifre sia esponenziale dell'ordine di $2^{k \log n}$. Allora il numero di operazioni richieste dall'algoritmo per un numero di $2 \log n$ cifre è dell'ordine di $2^{k \log n} \cdot 2^{k \log n}$. In altre parole, raddoppiando il numero di cifre dell'input, il numero di operazioni richieste si moltiplica per un fattore $2^{k \log n}$ che *dipende in modo crescente dalla grandezza dell'input.*

6. *(complessità delle operazioni aritmetiche sugli interi)* Siano n ed m interi rispettivamente di $\log n$ e $\log m$ cifre binarie.
   (a) Verificare che $n + m$ richiede $\mathcal{O}(\log n + \log m)$ operazioni.
   (b) Verificare che $n - m$ richiede $\mathcal{O}(\log n + \log m)$ operazioni.
   (c) Verificare che $n \cdot m$ richiede $\mathcal{O}(\log n \cdot \log m)$ operazioni.
   (d) Verificare che n^a richiede $\mathcal{O}(\log^2 n \cdot 2^{\log a})$ operazioni.
   (e) Verificare che $m : n$ richiede $\mathcal{O}(\log m \log n)$ operazioni.

Sol. (a)&(b) Siano $n = (a_{k-1}a_{k-2}\dots a_1a_0)_2$ e $m = (b_{h-1}b_{h-2}\dots b_1b_0)_2$, con $k \sim \log n$, $h \sim \log m$ e $k \geq h$. La somma

$$\begin{array}{r} a_k a_{k+1} \dots\dots a_1 a_0 + \\ b_h b_{h+1} \dots b_1 b_0 \end{array}$$

è ottenuta da al più h somme e k riporti. Dunque la complessità di $n+m$ è $\mathcal{O}(\log n + \log m)$. Per quanto osservato nell'Esercizio 4, lo stesso vale per $n-m$.
(c) Il prodotto $n \cdot m$

$$\begin{array}{r} a_{k-1}a_{k-2}\ldots\ldots a_1a_0\times \\ b_{h-1}b_{h-2}\ldots b_1b_0 = \\ ---------- \\ l_{k-1}l_{k-2}\ldots\ldots l_1l_0 \\ m_{k-1}m_{k-2}\ldots\ldots m_1m_0- \\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots - \ - \end{array}$$

è ottenuto sommando h righe di lunghezza k, ognuna traslata di un bit verso sinistra rispetto alla precedente. Queste righe sono date dagli h prodotti della stringa $a_k a_{k+1}\ldots\ldots a_1a_0$ per $b_0, b_1, \ldots, b_{h-1}$. Ogni riga richiede $\mathcal{O}(\log n)$ operazioni, sommarne due richiede $\mathcal{O}(2\log n)$ operazioni, sommarle tutte richiede $\mathcal{O}(\log m \cdot 2\log n)$ operazioni. In totale la complessità del prodotto $n \cdot m$ risulta

$$\mathcal{O}(\log m \cdot \log n + \log m \cdot 2\log n) \sim \mathcal{O}(\log n \cdot \log m).$$

(d) Scriviamo a in forma binaria

$$a = (\epsilon_\mu \epsilon_{\mu-1}\ldots\epsilon_0)_2, \quad \epsilon_i \in \{0,1\}.$$

Poiché $a = \sum_{i=0}^{\mu} \epsilon_i 2^i$, abbiamo

$$n^a = n^{\sum_{i=0}^{\mu}\epsilon_i 2^i} = \prod_{i=0}^{\mu} n^{\epsilon_i 2^i} = \prod_{i=0,\epsilon_i\neq 0}^{\mu} n^{2^i}.$$

Un monomio n^{2^i} è ottenuto mediante i quadrati successivi

$$n, \quad n^2 = n\cdot n, \quad n^{2^2} = n^2\cdot n^2, \quad n^{2^3} = n^{2^2}\cdot n^{2^2}, \quad \ldots.$$

Osserviamo che se n ha $k \sim \log n$ cifre binarie, $n^2 = n\cdot n$ ne ha $2k$, ed n^{2^i} ne ha $2^i k$. Il monomio n^{2^i} è ottenuto dal monomio precedente $n^{2^{i-1}}$ mediante una sola quadratura. Infine nel calcolare il monomio n^{2^μ} calcoliamo anche tutti i monomi precedenti. Dunque il calcolo di tutti i monomi n^{2^i}, per $i = 1,\ldots,\mu$ richiede all'incirca $(1+4+\ldots+2^{2(\mu-1)}\log^2 n$ operazioni, ossia ha complessità

$$\mathcal{O}(2^{2(\mu-1)}\log^2 n).$$

Adesso dobbiamo moltiplicare tutti i monomi fra loro. Nel valutare la complessità di questo prodotto dobbiamo tener conto che ad ogni passo cresce il numero di cifre dei fattori. I prodotti

$$n\cdot n^2, \quad (n\cdot n^2)\cdot n^{2^2}, \quad (n\cdot n^2\cdot n^{2^2})\cdot n^{2^3}, \ \ldots, \ (n\cdot n^2\cdot\ldots\cdot n^{2^{\mu-1}})\cdot n^{2^\mu}$$

richiedono rispettivamente circa

$$\log n\cdot 2\log n, \quad (1+2)\log n\cdot 2^2\log n, \quad (1+2+2^2)\log n\cdot 2^{2^3}\log n, \ldots, \ (1+2+\ldots+2^{\mu-1})\log n\cdot 2^\mu\log n$$

operazioni. In totale la complessità di n^a risulta

$$\mathcal{O}(2^{2(\mu-1)}\log^2 n + 2^{2\mu-1}\log^2 n) \sim \mathcal{O}(2^{2\mu}\log^2 n) = \mathcal{O}(2^{2\log a}\log^2 n), \qquad \mu \sim \log a.$$

Osserviamo che la complessità di questo calcolo è *polinomiale* in $\log n$ ma *esponenziale* in $\log a$.

(e) Per stimare la complessità della divisione con resto $n : m$, cioè $n = qm + r$, osserviamo che per ottenere q ed r facciamo circa $\log q$ sottrazioni fra stringhe di lunghezza circa $\log m$:

$$\begin{array}{l} a_{k-1}a_{k-2}\ldots\ldots a_1a_0 \quad : \quad b_{h-1}b_{h-2}\ldots b_1b_0 \quad = \quad c_pc_{p-1}\ldots \\ b_{h-1}b_{h-2}\ldots b_1b_0 \\ \qquad * * \cdots * * \\ \qquad b_{h-1}b_{h-2}\ldots b_1b_0 \\ \qquad\quad \ldots \end{array}$$

Quindi è data da $\mathcal{O}(\log n \log q)$. D'altra parte $\log q$ può essere maggiorato con $\log n$, da cui segue che la complessità della divisione $n : m$ è data da $\mathcal{O}(\log n \log m)$.

7. Sia $b \in N$ un intero positivo fissato. Sia x un intero. Determinare la complessità del calcolo dell'espressione di x in base b
$$x = (a_n a_{n-1} \ldots a_1 a_0)_b.$$

Sol. L'espressione di x in base b si ottiene mediante divisioni successive per b: dividendo x per b, poi dividendo il quoziente così ottenuto per b e così via fino a che non si ottiene un quoziente minore di b.

$$x = q_0 b + r_0, \quad q_0 = q_1 b + r_1, \quad q_1 = q_2 b + r_2, \quad \ldots, \quad q_{k-1} = q_k b + r_k, \qquad 0 < q_k < b$$

L'espressione di x in base b risulterà allora $x = (r_k r_{k-1} \ldots r_1 r_0)_b$. La complessià di questo calcolo è data dal numero di divisioni necessarie, ossia $\log_b x = \frac{\log x}{\log b}$, per la complessità di una singola divisione, che può essere maggiorata da $\log x \log b$. In totale

$$\mathcal{O}(\log_b x \cdot \log x \log b) \sim \mathcal{O}(\frac{\log x}{\log b} \cdot \log x \log b) \sim \mathcal{O}(\log^2 x).$$

8. *(complessità dell'algoritmo di Euclide)* Siano m ed n interi rispettivamente di $\log m$ e $\log n$ cifre binarie. Verificare che l'algoritmo di Euclide per determinare $\gcd(m, n)$, il massimo comun divisore fra m ed n, ha una complessità dell'ordine di $\mathcal{O}(\log m \cdot \log^2 n)$.

Sol. Siano n ed m due interi, con $n > m > 0$. L'algoritmo di Euclide consiste in una succesione di divisioni con resto che alla fine producono il massimo comun divisore $\gcd(n, m)$ fra n ed m.

$$\begin{array}{rll} m &= r_0 & \\ n &= q_0 m + r_1, & 0 < r_1 < m \\ r_0 &= q_1 r_1 + r_2, & 0 < r_2 < r_1 \\ r_1 &= q_2 r_2 + r_3, & 0 < r_3 < r_2 \\ \vdots &= \quad \vdots & \\ r_{k-1} &= q_k r_k + r_{k+1}, & 0 < r_{k+1} < r_k \\ \vdots &= \quad \vdots & \end{array}$$

Questa successione di divisioni produce una successione decrescente di resti

$$0 \leq \ldots r_{k+1} < r_k < r_{k-1} < \ldots r_1 < r_0 = m.$$

Il massimo comun divisore $gcd(n, m)$ è l'ultimo resto positivo di questa successione: se $r_{k+1} = 0$, il massimo comun divisore cercato è r_k.

La prima divisione richiede al più $\log n \log m$ operazioni, la seconda $\log m \log r_1$ operazioni, la terza $\log r_1 \log r_2$, etc...
Ad ogni modo il numero di operazioni richieste da ogni singola divisione può essere maggiorato con $\log n \log m$.
Resta da stimare quante divisioni sono necessarie ad ottenere il massimo comun divisore $\gcd(n, m)$.

Osservazione. Nella peggiore delle ipotesi, se i resti calassero di uno, per arrivare al massimo comun divisore, sarebbero necessarie $m = 2^{\log m}$ divisioni, e la complessità dell'algoritmo risulterebbe esponenziale in $\log m$. Il prossimo lemma implica che sono sufficienti $\log m$ divisioni, per cui l'algoritmo è effettivamente polinomiale in $\log n$ e $\log m$.

Lemma. *Al più ogni due passi dell'algoritmo il resto si dimezza:* $r_{k+2} \leq \frac{1}{2} r_k$.

Dim. Se $r_{k+1} \leq \frac{1}{2} r_k$, allora a maggior ragione $r_{k+2} < r_{k+1} \leq \frac{1}{2} r_k$. Se $r_{k+1} > \frac{1}{2} r_k$, abbiamo innanzitutto che nell'equazione $r_k = q_{k+1} r_{k+1} + r_{k+2}$ il quoziente $q_{k+1} = 1$. Dopodiché $r_{k+2} \leq \frac{1}{2} r_k$, come richiesto.

Il lemma implica che per ottenere il massimo comun divisore $\gcd(m, n)$ sono necessarie al più $2 \log m$ divisioni e che in totale la complessità del $\gcd(n, m)$ è

$$\mathcal{O}(2 \log m \cdot \log n \log m) \sim \mathcal{O}(\log n \log^2 m).$$

9. Siano $n > m > 0$ interi e sia $d = \gcd(n, m)$. L'algoritmo di Euclide esteso produce interi x, y che soddisfano l'equazione diofantea $nx + my = d$. Mostrare che $|x| \leq m/d$ e $|y| \leq n/d$.

Sol. Seguendo i passi dell'algoritmo di Euclide esteso, troviamo una successione di identità

$$\begin{aligned}
1 \cdot n &+ 0 \cdot m = n \\
0 \cdot n &+ 1 \cdot m = m \\
1 \cdot n &+ (-q_0) \cdot m = r_1 \\
-q_1 \cdot n &+ (1 + q_0 q_1) \cdot n = r_2 \\
\vdots &\quad \vdots \quad \vdots \\
a_{k-1} \cdot n &+ b_{k-1} \cdot m = r_{k-1} \\
a_k \cdot n &+ b_k \cdot m = r_k \\
a_{k+1} \cdot n &+ b_{k+1} \cdot m = r_{k+1} \\
\vdots &\quad \vdots \quad \vdots
\end{aligned}$$

dove $r_0 > r_1 > \ldots > r_{k-1} > r_k > r_{k+1} \ldots > 0$ è la successione dei resti delle divisioni successive dell'algoritmo (vedi esercizio 8) e i coefficienti sono dati dalle relazioni

$$a_{k+1} = a_{k-1} - q_k a_k, \qquad b_{k+1} = b_{k-1} - q_k b_k, \qquad q_k \geq 1. \tag{R}$$

Dalle relazioni (R) seguono le seguenti proprietà dei coefficienti:

(a) gli a_k e b_k cambiano di segno ogni volta che k cresce di una unità.
(b) $|a_{k+1}| > |a_k|$ e $|b_{k+1}| > |b_k|$, ossia gli a_k e b_k crescono in valore assoluto al crescere di k.
(c) $\gcd(a_k, b_k) = 1$, per ogni k.

Dim.(a): Abbiamo $a_1 > 0$ e $a_2 < 0$. Dalle relazioni (R), segue che se $a_{k-1} > 0$ e $a_k < 0$, allora $a_{k+1} > 0$. Stesso ragionamento per i b_k.

(b): Se $a_{k-1} > 0$ e $a_k < 0$, dalle relazioni (R) segue che $|a_{k+1}| \geq -q_k a_k = |q_k a_k| > |a_k|$.

(c): Cominciamo col dimostrare che $\det \begin{pmatrix} a_k & b_k \\ a_{k+1} & b_{k+1} \end{pmatrix} = \pm 1$, per ogni $k \geq 0$.

Infatti, per $k = 0$, vale $\det \begin{pmatrix} 0 & 1 \\ 1 & (-q_0) \end{pmatrix} = -1$; inoltre se $\det \begin{pmatrix} a_{k-1} & b_{k-1} \\ a_k & b_k \end{pmatrix} = \pm 1$, allora

$$\det \begin{pmatrix} a_k & b_k \\ a_{k+1} & b_{k+1} \end{pmatrix} = a_k b_{k-1} - b_k a_{k-1} = \pm 1.$$

Per il teorema di Bezout, la relazione $a_k b_{k-1} - b_k a_{k-1} = \pm 1$ implica che $\gcd(a_k, b_k) = 1$, come richiesto.

Ritorniamo adesso alle identità dell'algoritmo di Euclide esteso, in particolare alle ultime due, che saranno del tipo

$$\begin{aligned} a_k \cdot n &+ b_k \cdot m = d \\ a_{k+1} \cdot n &+ b_{k+1} \cdot m = 0. \end{aligned}$$

L'ultima è equivalente all'equazione

$$a_{k+1} \cdot \frac{n}{d} + b_{k+1} \cdot \frac{m}{d} = 0,$$

dove $\gcd(a_{k+1}, b_{k=1}) = \gcd(\frac{n}{d}, \frac{m}{d}) = 1$. Ne segue immediatamente che a_{k+1} divide $\frac{m}{d}$ e b_{k+1} divide $\frac{n}{d}$, da cui la tesi.

10. Sia $n \in N$ un intero positivo fissato. Sia x un intero. Determinare la complessità del calcolo di $\bar{x} \in \mathbb{Z}_n$, ossia della classe resto di x modulo n.

Sol. La complessità del calcolo di $\bar{x}$ modulo n è quivalente a quella della divisione con resto $x = qn + r$. Infatti per definizione $\bar{x} = r$. Perciò la complessità risulta $\mathcal{O}(\log x \log n)$.

11. *(complessità delle operazioni aritmetiche in $\mathbb{Z}_n$)* Siano dati $\bar{x}$ ed $\bar{y}$ in $\mathbb{Z}_n$.
    (a) Verificare che $\bar{x} + \bar{y}$ richiede $\mathcal{O}(\log n)$ operazioni.
    (b) Verificare che $\bar{x} - \bar{y}$ richiede $\mathcal{O}(\log n)$ operazioni.
    (c) Verificare che $\bar{x} \cdot \bar{y}$ richiede $\mathcal{O}(\log^2 n)$ operazioni.
    (d) Verificare che $\bar{x}^a$ richiede $\mathcal{O}(\log^2 n \cdot \log a)$ operazioni.
    (e) Verificare che $\bar{x}^{-1}$ richiede $\mathcal{O}(\log^3 n)$ operazioni.

Sol. Osserviamo innanzituto che per valutare la complessità delle varie operazioni modulo n

$$\bar{x} + \bar{y}, \quad \bar{x} - \bar{y}, \quad \bar{x} \cdot \bar{y}, \quad \bar{x}^a, \quad \bar{x}^{-1}$$

conviene escludere dalla discussione il calcolo iniziale delle classi $\bar{x}$ e $\bar{y}$ modulo n. Infatti se x e y sono molto grandi, la complessità di tale calcolo oscura il resto. Osserviamo inoltre che in un algoritmo su $\mathbb{Z}_n$ è necessario fare la riduzione modulo n ad ogni passo, per tenere la grandezza dei numeri sotto controllo.

(a) I numeri $\bar{x}$ e $\bar{y}$ sono entrambi minori di n; la somma $\bar{x} + \bar{y}$ richiede $\mathcal{O}(\log n)$ operazioni. Poiché $\bar{x} + \bar{y}$ è minore di $2n$, il calcolo della classe resto $\overline{\bar{x} + \bar{y}}$ modulo n richiede altre $\mathcal{O}(\log n)$ operazioni. (Perché non $\mathcal{O}(\log^2 n)$?? siccome $\bar{x} + \bar{y} < 2n$, in questo caso la classe $\overline{\bar{x} + \bar{y}}$ modulo n si ottiene direttamente sottraendo n da $\bar{x} + \bar{y}$). In totale abbiamo

$$\mathcal{O}(\log n + \log n) \sim \mathcal{O}(\log n).$$

(b) Ricordiamo che $\bar{x} - \bar{y} = \bar{x} + (-\bar{y})$ e che $-\bar{y} = \overline{-\bar{y} + n} \in \{0, \ldots, n-1\}$. Dopodiché si ragiona come nel caso precedente.
(c) La moltiplicazione $\bar{x} \cdot \bar{y}$ richiede $\mathcal{O}(\log^2 n)$ operazioni. Poiché il prodotto $\bar{x} \cdot \bar{y}$ è minore di n^2, il calcolo della classe resto $\overline{\bar{x} \cdot \bar{y}}$ modulo n richiede altre $\mathcal{O}(2 \log n \log n) \sim \mathcal{O}(\log^2 n)$ operazioni. In totale abbiamo

$$\mathcal{O}(\log^2 n + \log^2 n) \sim \mathcal{O}(\log^2 n).$$

(d) Scrivendo a in forma binaria $a = (\epsilon_\mu \epsilon_{\mu-1} \ldots \epsilon_0)_2$, abbiamo

$$\bar{x}^a = \prod_{i=0, \epsilon_i \neq 0}^{\mu} \bar{x}^{2^i}.$$

Il monomio $\bar{x}^{2^\mu}$, insieme a tutti i monomi precedenti, è ottenuto mediante μ quadrati successivi e relativa riduzione modulo n

$$\bar{x}, \quad \overline{\bar{x} \cdot \bar{x}}, \quad \overline{\overline{\bar{x} \cdot \bar{x}} \cdot \overline{\bar{x} \cdot \bar{x}}}, \quad \ldots$$

e richiede in totale $\mathcal{O}(\mu \log^2 n)$ operazioni. Il prodotto fra i μ monomi richiede al più $\mathcal{O}(\mu \log^2 n)$ operazioni. In totale

$$\mathcal{O}(\mu \log^2 n + \mu \log^2 n) \sim \mathcal{O}(\mu \log^2 n) = \mathcal{O}(\log a \log^2 n), \qquad \mu \sim \log a.$$

Osserviamo che modulo n, il calcolo di $\bar{x}^a$ è polinomiale in sia in $\log n$ che in $\log a$ (confronta con l'Esercizio 6).
(e) Una classe $\bar{x} \in \mathbb{Z}_n$ ammette inverso moltiplicativo se e solo se $\gcd(x, n) = 1$. Al termine dell'algoritmo di Euclide per calcolare $\gcd(x, n) = 1$ troviamo una relazione del tipo

$$x\alpha + n\beta = 1, \quad \alpha, \beta \in \mathbb{Z}. \tag{$*$}$$

Per definizione $\bar{\alpha}$ è l'inverso moltiplicativo di $\bar{x}$ in $\mathbb{Z}_n$. Dunque la complessità del calcolo di $\bar{x}^{-1}$ è quella dell'algoritmo di Euclide, cioè $\mathcal{O}(\log^3 n)$, più quella del calcolo di $\bar{a}$ modulo n che è $\mathcal{O}(\log a \log n)$.
Dal risultato dell'esercizio 9, per $d = \gcd(x, n) = 1$, segue che $|\alpha| \leq n$. Dunque la complessità del calcolo di $\bar{a}$ modulo n è maggiorata da $\mathcal{O}(\log^2 n)$. e la complessità del calcolo di $\bar{x}^{-1}$ in $\mathbb{Z}_n$ risulta

$$\mathcal{O}(\log^3 n + \log^2 n) \sim \mathcal{O}(\log^3 n).$$

12. Determinare la complessità del test di primalità di Miller-Rabin usando 3 basi.

Sol. Analizziamo le varie fasi dell'algoritmo per determinare se un intero n è pseudoprimo rispetto ad una base a (vedi nota 2):

- scrivere $n - 1 = m2^k$, con m dispari (k divisioni di $n - 1$ per 2): $\mathcal{O}(k \log n)$ operazioni;
- calcolare $b := a^m$ modulo n. $\mathcal{O}(\log m \log^2 n)$ operazioni;
- calcolare al più k quadrati $b^2,\ b^{2^2},\ \ldots, b^{2^k}$ modulo n. $\mathcal{O}(k \log^2 n)$ operazioni;

Poiché possiamo maggiorare $k \leq \log n$ e $\log m \leq \log n$, per *una* base la complessità totale risulta

$$\mathcal{O}(\log^3 n).$$

Per *tre* basi, risulta

$$\mathcal{O}(3 \log^3 n) \sim \mathcal{O}(\log^3 n).$$

Anche ripetendo l'algoritmo per $\log n$ basi diverse, la complessità resta polinomiale $\mathcal{O}(\log^4 n)$. Ricordiamo che la probabilità che un intero n sia a-pseudoprimo per N basi distinte, senza essere primo, può essere maggiorata da $\frac{1}{4^N}$.

13. Sia n un intero di 100 cifre decimali. Supponiamo che il test di Miller-Rabin su n usando 3 basi richieda un tempo T. Quanto tempo richiede lo stesso test su un numero m di 200 cifre decimali? E su un numero N di 1000 cifre decimali?

Sol. Su un numero m di 200 cifre decimali richiede un tempo $2^3 T = 8T$, su un numero m di 1000 cifre decimali richiede un tempo $10^3 T = 1000T$.