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

La *complessità di un algoritmo* è il numero di operazioni elementari (sui singoli bit) da esso richieste in funzione del numero di cifre degli elementi dell'input. Siamo interessati al comportamento asintotico di questa funzione, al crescere del numero di cifre degli elementi dell'input.

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 degli elementi dell'input; si dice *subesponenziale* se la sua complessità è una funzione subesponenziale nel numero di cifre degli elementi dell'input.

Esempio. Supponiamo che l'input di un dato algoritmo sia costituito da un numero intero n.
- La sua complessità è polinomiale se è una funzione che al crescere di n ha lo stesso ordine di infinito di $\log^k n$, per un $k \in \mathbb{N}$. In tal caso diciamo che è $\mathcal{O}(\log^k n)$.
- La sua complessità è esponenziale se è una funzione che al crescere di n ha lo stesso ordine di infinito di $n^k = 2^{k \log n}$, per un $k \in \mathbb{N}$. In tal caso diciamo che è $\mathcal{O}(2^{k \log n})$.
- La sua complessità è subesponenziale se è una funzione che al crescere di n ha ordine di infinito inferiore a $n^k = 2^{k \log n}$, per ogni $k > 0$. Ogni funzione polinomiale è subesponenziale. Una funzione subesponenziale, non polinomiale di $\log n$ è ad esempio $2^{\sqrt{\log n}}$.

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 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 $n \cdot m$

$$\begin{array}{r} a_{k-1}a_{k-2} \dots\dots a_1 a_0 \times \\ b_{h-1}b_{h-2} \dots b_1 b_0 = \\ - - - - - - - - - - - - - \\ l_{k-1}l_{k-2} \dots\dots l_1 l_0 \\ m_{k-1}m_{k-2} \dots\dots m_1 m_0 - \\ \dots\dots\dots\dots\dots\dots - \ - \end{array}$$

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 $a_k a_{k+1} \dots\dots a_1 a_0$ per b_0, b_1, …, 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

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

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

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

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 x^{2^i} è ottenuto dal monomio precedente $x^{2^{i-1}}$ mediante una sola quadratura. Infine nel calcolare il monomio x^{2^μ} calcoliamo anche tutti i monomi precedenti. Dunque il calcolo di tutti i monomi x^{2^i}, per $i = 1, \dots, \mu$ richiede all'incirca $(1 + 4 + \dots + 2^{\mu-1}) \log^2 n$ operazioni, ossia ha complessità

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

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

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

richiedono rispettivamente circa

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

operazioni. In totale la complessità di n^a risulta

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

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 $\log m$:

$$\begin{array}{l} a_{k-1}a_{k-2} \dots\dots a_1 a_0 \quad : \quad b_{h-1}b_{h-2} \dots b_1 b_0 \quad = \quad c_p c_{p-1} \dots \\ b_{h-1}b_{h-2} \dots b_1 b_0 \\ \qquad * * \cdots * * \\ \qquad b_{h-1}b_{h-2} \dots b_1 b_0 \\ \qquad\quad \dots \end{array}$$

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

$$x = (a_n a_{n-1} \dots a_1 a_0)_b.$$

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

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

Questa successione di divisioni produce una successione decrescente di resti

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

Il massimo comun divisore $gcd(m, n)$ 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 $\log m \log n$ operazioni, la seconda $\log n \log r_1$ 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 $\log m \log n$.
Resta da stimare quante divisioni sono necessarie ad ottenere il massimo comun divisore $\gcd(m, n)$.

Osservazione. *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.

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

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

9. 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 $\mathcal{O}(\log x \log n)$.

10. *(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 $-\bar{y}=\overline{-\bar{y}+n}\in\{0,\dots,n-1\}$. 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 $a=(\epsilon_\mu\epsilon_{\mu-1}\dots\epsilon_0)_2$, 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

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

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) Se $\gcd(\bar{x},n)=1$ allora $\bar{x}$ è invertibile in $\mathbb{Z}_n$. Al termine dell'algoritmo di Euclide per calcolare $\gcd(\bar{x},n)=1$ troviamo una relazione del tipo

$$a\bar{x}+bn=1, \quad a,b\in\mathbb{Z}. \tag{$*$}$$

Per definizione $\bar{a}$ è l'inverso 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 $\mathcal{O}(\log a\log n)$.

Verifichiamo adesso che la complessità del calcolo di $\bar{a}$ modulo n è maggiorata da $\mathcal{O}(\log^2 n)$. Supponiamo che $\gcd(a,n)=1=r_{k+1}$. Seguendo i vari passi dell'algoritmo di Euclide per il calcolo di $\gcd(\bar{x},n)$

$$\begin{aligned}
1\cdot\bar{x} \quad &+ \quad 0\cdot n = \bar{x}\\
0\cdot\bar{x} \quad &+ \quad 1\cdot n = n\\
1\cdot\bar{x} \quad &- \quad q_0\cdot n = r_1\\
-q_1\cdot\bar{x} \quad &+ (1+q_0q_1)\cdot n = r_2\\
(1+q_1q_2)\cdot\bar{x} \quad &- (q_0+q_2+q_0q_1q_2)\cdot n = r_3\\
\vdots \quad &\vdots \quad \vdots
\end{aligned}$$

si vede che il coefficiente a in (*) ha grossomodo la grandezza del prodotto dei quozienti successivi $q_0 < \frac{n}{m} = \frac{n}{r_0},\ q_1 < \frac{r_0}{r_1}, \ldots, q_k < \frac{r_{k-1}}{r_k}$:

$$a \sim q_0 q_1 q_2 \ldots q_k \ < \ \frac{n}{r_0}\frac{r_0}{r_1} \ldots \frac{r_{k-1}}{r_k} = \frac{n}{r_k}, \quad r_k > 1.$$

Quindi $a < n$ e $\log a \leq \log n$. In conclusione la complessità del calcolo di $\bar{x}^{-1}$ in $\mathbb{Z}_n$ è

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

11. 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 $\frac{1}{4^N}$.

12. 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 $10^3 T = 1000T$.