ANALISI DELLA COMPLESSITÀ DEGLI ALGORITMI

Definizioni di base

Algoritmo: procedura computazionale ben definita che prende valori in **input** e produce valori in **output**.

Un algoritmo è uno strumento per risolvere un **problema computazionale**, che specifica la relazione che deve valere tra input e output.

Esempio: problema dell'ordinamento di una sequenza di numeri in senso non decrescente:
Input: una sequenza di n numeri $\langle a_1, a_2, ..., a_n \rangle$;
Output: una permutazione $\langle a'_1, a'_2, ..., a'_n \rangle$ di $\langle a_1, a_2, ..., a_n \rangle$ tale che $a'_1 \leq a'_2 \leq ... \leq a'_n$.
Istanza del problema di ordinamento:
sequenza di input $\langle 31, 41, 59, 26, 41, 58 \rangle$;
Soluzione a questa istanza del problema:
sequenza di output $\langle 26, 31, 41, 41, 58, 59 \rangle$.

Istanza di un problema computazionale: insieme di dati di input (che soddisfano eventuali vincoli)

Un algoritmo che risolve il problema computazionale P è **corretto** se per ogni istanza X del problema P l'algoritmo si ferma e fornisce un output Y corretto, cioè tale che tra X e Y vale la relazione specificata da P.

Un buon algoritmo fa esattamente quello che deve fare usando una quantità minima di risorse.

Ordinamento di un array di 1.000.000 di numeri

hardware:	supercalcolatore	personal computer
linguaggio:	macchina	di alto livello
compilatore:		non efficiente
programmatore:	esperto	medio
algoritmo:	insertion sort	merge sort
tempo:	5.56 ore	16.67 minuti

Pseudocodifica di Algoritmi: Insertion sort

Input: un array $A[1..n]$ di numeri ($n = length[A]$)

Ordinamento in loco

```
INSERTION-SORT (A)
1. for j <- 2 to length[A]
2.     do   key <- A[j]
3.          ;; si inserisce key nella sequenza ordinata A[1,...,j-1]
            ;; spostando a destra gli elementi maggiori di key
4.          i <- j-1
5.          while i>0 e A[i]>key
6.                do A[i+1] <- A[i]
7.                   i <- i-1
8.          A[i+1] <- key
```

j=2	5	**2**	4	6	1	3
j=3	2	5	**4**	6	1	3
j=4	2	4	5	**6**	1	3
j=5	2	4	5	6	**1**	3
j=6	1	2	4	5	6	**3**
	1	2	3	4	5	6

Convenzioni per lo Pseudocodice

1. Le rientranze indicano la struttura dei blocchi
2. Costrutti condizionali: if-then-else
   Cicli: while, for, repeat
3. Commenti: ;;
4. Assegnazioni multiple $i \leftarrow j \leftarrow e$ ($i \leftarrow e;\ j \leftarrow e$)
5. Variabili locali alle procedure (a meno che non sia indicato esplicitamente)
6. L'indicizzazione degli array inizia da 1.
   Se A è un array e i un indice, $A[i]$ è l'elemento in posizione i.
   $A[1, ..., i]$ indica il sottoarray di A: $A[1], A[2], ..., A[i]$.
7. Dati compositi organizzati in oggetti, stutturati in attributi o campi.
   Accesso al campo: $campo[oggetto]$ Esempio: $length[A]$
8. Una variabile che rappresenta un oggetto è trattata come un *puntatore* (riferimento) all'oggetto.
   Con l'assegnazione $y \leftarrow x$ si ha $campo[y] = campo[x]$
   Se si modifica un campo di x: $campo[x] \leftarrow 3$, si ha anche $campo[y] = 3$.
9. NIL: puntatore che non si riferisce al alcun oggetto.
10. I parametri di una procedura sono sempre passati **per valore**: la procedura chiamata riceve una copia dei parametri, se si modifica il valore di un parametro, la modifica non ha effetto all'esterno.
    Se il parametro è un oggetto viene passata una copia del puntatore all'oggetto; non sono copiati i campi: l'assegnazione $A[i+1] \leftarrow key$ è visibile all'esterno.

Analisi di Algoritmi

Obiettivo: prevedere le *risorse* richieste dall'algoritmo

- tempo di calcolo
- memoria
- traffico generato su rete
- trasferimento dati da/su disco
- ....

Analisi basata su un modello di calcolo: un modello della tecnologia usata per realizzare l'algoritmo.
RAM (Random Access Machine): mono-processore, istruzioni eseguite in modo sequenziale.

Dimensione dell'input

Il tempo di esecuzione di un algoritmo dipende dall'input: dalla sua dimensione e dalla sua struttura.

In generale il tempo cresce al crescere dell'input:

Tempo di esecuzione misurato in termini delle dimensioni dei dati di ingresso: espresso come funzione, in modo da riassumere le prestazioni su tutti i dati di ingresso.

La nozione di **dimensione dell'input** dipende dal problema:

- Ordinamento: numero di elementi da ordinare
- Risoluzione di n equazioni lineari in n incognite: n^2 coefficienti
- Operazioni su liste: lunghezza della lista
- Problemi numerici: numero di bit necessari per rappresentare l'input
- Valutazione di un polinomio in un punto: grado del polinomio

Tempo di esecuzione

Tempo di esecuzione di un algoritmo per un particolare input

$=$

numero di operazioni elementari eseguite per il calcolo dell'output.

$$T(n)$$

Tempo di esecuzione dell'algoritmo su input generico di dimensione n

Cos'è un'operazione elementare?

Ipotesi semplificativa per rendere la nozione indipendente dalla macchina:

> per eseguire una linea (o istruzione) di pseudocodice è richiesto tempo costante:
> per eseguire la riga i si impiega tempo c_i

Esempio - Insertion Sort (I)

	INSERTION-SORT (A)	*costo*	n^o *di volte*
1.	`for j <- 2 to length[A]`	c_1	n
2.	`  do key <- A[j]`	c_2	$n-1$
3.	`  ;; si inserisce A[j]`	0	$n-1$
4.	`  i <- j-1`	c_4	$n-1$
5.	`  while i>0 e A[i]>key`	c_5	$\Sigma_{j=2}^{n} t_j$
6.	`  do A[i+1] <- A[i]`	c_6	$\Sigma_{j=2}^{n}(t_j-1)$
7.	`  i <- i-1`	c_7	$\Sigma_{j=2}^{n}(t_j-1)$
8.	`  A[i+1] <- key`	c_8	$n-1$

dove t_j è il numero di volte che viene eseguito il test del ciclo while alla riga 5 per quel valore di j (dipende dalla struttura dell'input).

1. j viene confrontato con $length[A]$ per $j = 2, ..., n, n+1$: l'ultimo confronto determina l'uscita dal ciclo.

2-4. Il ciclo viene eseguito $n-1$ volte (per $j = 2, ..., n$), quindi le istruzioni alle righe 2, 3 e 4 vengono eseguite $n-1$ volte.

5. Per ogni $j = 2, ..., n$ (per $n-1$ volte) viene eseguito il test del ciclo while un numero di volte t_j dipendente dalla struttura dell'input. L'ultimo test determina l'uscita dal ciclo.

Insertion Sort (II)

	INSERTION-SORT (A)	*costo*	n^o *di volte*
1.	`for j <- 2 to length[A]`	c_1	n
2.	`  do key <- A[j]`	c_2	$n-1$
3.	`  ;; si inserisce A[j]...`	0	$n-1$
4.	`  i <- j-1`	c_4	$n-1$
5.	`  while i>0 e A[i]>key`	c_5	$\Sigma_{j=2}^{n} t_j$
6.	`  do A[i+1] <- A[i]`	c_6	$\Sigma_{j=2}^{n}(t_j-1)$
7.	`  i <- i-1`	c_7	$\Sigma_{j=2}^{n}(t_j-1)$
8.	`  A[i+1] <- key`	c_8	$n-1$

6-7. Il ciclo while viene eseguito $t_j - 1$ volte, per ogni $j = 2, ..., n$ (quindi per $n-1$ volte).

8. L'ultima istruzione viene eseguita per ogni $j = 2, ..., n$ (quindi per $n-1$ volte).

Costo dell'algoritmo = somma dei costi

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \Sigma_{j=2}^{n} t_j + c_6 \Sigma_{j=2}^{n}(t_j-1) + c_7 \Sigma_{j=2}^{n}(t_j-1) + c_8(n-1)$$

Insertion Sort - Il caso migliore

l'array è già ordinato

$$\begin{aligned} T(n) = \; & c_1 n + c_2(n-1) + c_4(n-1) + c_5 \Sigma_{j=2}^{n} t_j + \\ & c_6 \Sigma_{j=2}^{n}(t_j - 1) + c_7 \Sigma_{j=2}^{n}(t_j - 1) + c_8(n-1) \end{aligned}$$

Il tempo di esecuzione dipende dalla struttura dell'input.

Per ogni j, si esce subito dal ciclo while perché $A[j-1]$ (che sarebbe $A[i]$) è minore o uguale di $A[j]$.

In questo caso $\Sigma_{j=2}^{n} t_j = n - 1$ e $\Sigma_{j=2}^{n}(t_j - 1) = 0$:

- Il passo `5.` viene eseguito $n - 1$ volte
- I passi `6.` e `7.` non vengono eseguiti

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \\ &= an + b \end{aligned}$$

per qualche costante a e b, dipendenti dai costi c_i.

Nel caso migliore $T(n)$ è una funzione **lineare** *di n.*

L'array è ordinato in senso inverso.

Per ogni $j = 2, ..., n$, $A[j]$ viene confrontato con ogni elemento del sottoarray $A[1, ..., j-1]$: $t_j = j$ (un confronto per ogni elemento di $A[1, ..., j-1]$ + uno per determinare l'uscita dal ciclo, quando $i = 0$).

Poiché:

$$\sum_{j=1}^{n} = \frac{n(n+1)}{2}$$

5.*diventa* $\Sigma_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ 6. *e* 7. *diventano* $\Sigma_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$

$$\begin{aligned} T(n) = \; & c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + \\ & c_6\frac{n(n-1)}{2} + c_7\frac{n(n-1)}{2} + c_8(n-1) \\ = \; & \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ & -(c_2 + c_4 + c_5 + c_8) \\ = \; & an^2 + bn + c \end{aligned}$$

Nel caso peggiore $T(n)$ *è una funzione* quadratica *di* n.

Insertion Sort - Tempo di esecuzione nel caso peggiore

Normalmente si prenderà in considerazione il tempo di esecuzione nel caso peggiore:

- Rappresenta un limite superiore al tempo di esecuzione per ogni input: $T(n) =$ massimo tempo di esecuzione su qualsiasi dato di dimensione n.
- Spesso il caso peggiore si verifica frequentemente.
- Il "caso medio" è spesso vicino al caso peggiore.

Ordine di Grandezza

Astrazioni (semplificazioni):

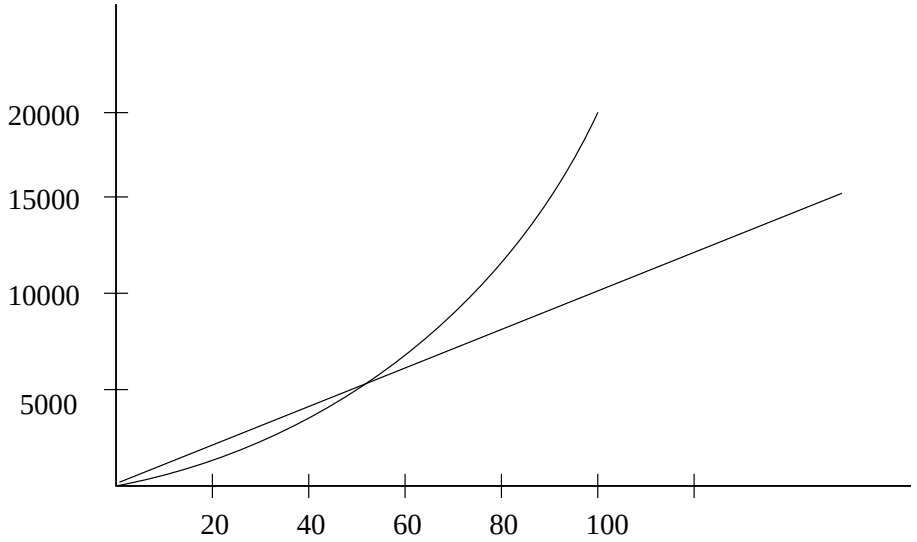
- si ignora il costo effettivo di ciascun comando: il numero di istruzioni macchina generate da un particolare compilatore per ogni istruzione "primitiva" e il numero medio di istruzioni macchina che un particolare calcolatore esegue nell'unità di tempo
- si ignora anche il costo astratto dei comandi:
  $T(n) = an^2 + bn + c$
- si considera soltanto l'ordine di grandezza di $T(n)$
  (analisi asintotica):

$T(n) = an + b$	lineare	$\Theta(n)$
$T(n) = an^2 + bn + c$	quadratico	$\Theta(n^2)$
$T(n) = c_0 n^k + c_1 n^{k-1} + ... + c_k$	polinomiale	$\Theta(n^k)$
$T(n) = a2^n$	esponenziale	$\Theta(2^n)$

Ordine di grandezza: tasso di crescita della funzione,
si ignorano i termini di ordine inferiore e i fattori moltiplicativi (coefficiente del termine principale).
Per input sufficientemente grandi essi sono trascurabili rispetto agli effetti della dimensione stessa dell'input.

Se $T_A(n)$ ha un ordine di grandezza inferiore di $T_B(n)$, l'algoritmo A si considera più efficiente dell'algoritmo B: per input sufficientemente grandi, A si comporterà meglio di B.

Per n che tende all'infinito, un algoritmo lineare è migliore di uno quadratico.



$$T_A(n) = 100 \times n \qquad T_B(n) = 2 \times n^2$$

Se:	$n < 50$:	$T_B(n) \leq T_A(n)$;	$T_B(n) = 5000, \quad T_A(n) = 5000$
	$n = 100$	$T_B(n) = 2 \times T_A(n)$	$T_B(n) = 20000, \quad T_A(n) = 10000$
	$n = 1000$	$T_B(n) = 20 \times T_A(n)$	$T_B(n) = 2000000, \quad T_A(n) = 100000$

Quale algoritmo è migliore dipende dalla dimensione massima dell'input che il programma deve elaborare

Ordine di grandezza delle funzioni: Notazioni asintotiche

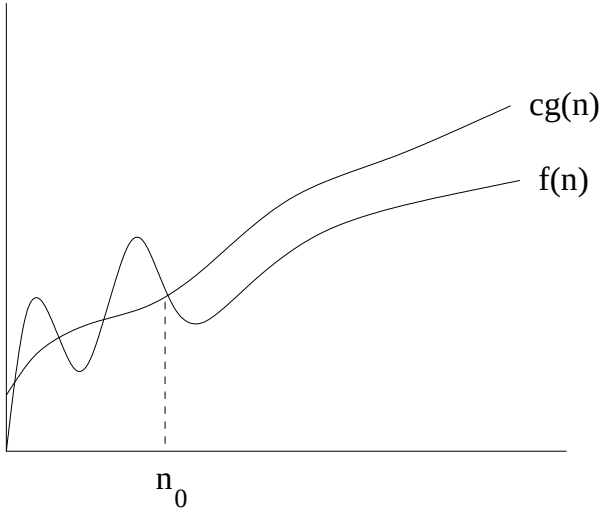
Funzioni $f(n)$ il cui dominio è $\mathbb{N}$.

Limite asintotico superiore: notazione O-grande

Insieme delle funzioni limitate superiormente (da un certo punto in poi)

$$O(g(n)) = \{f(n) : \text{esistono costanti positive } n_0 \text{ e } c \text{ tali che } 0 \leq f(n) \leq c \cdot g(n) \text{ per ogni } n \geq n_0\}.$$

$f(n) \in O(g(n))$ sse:
esistono costanti positive n_0 e c tali che:
$\forall n \geq n_0 : \quad 0 \leq f(n) \leq c \cdot g(n)$



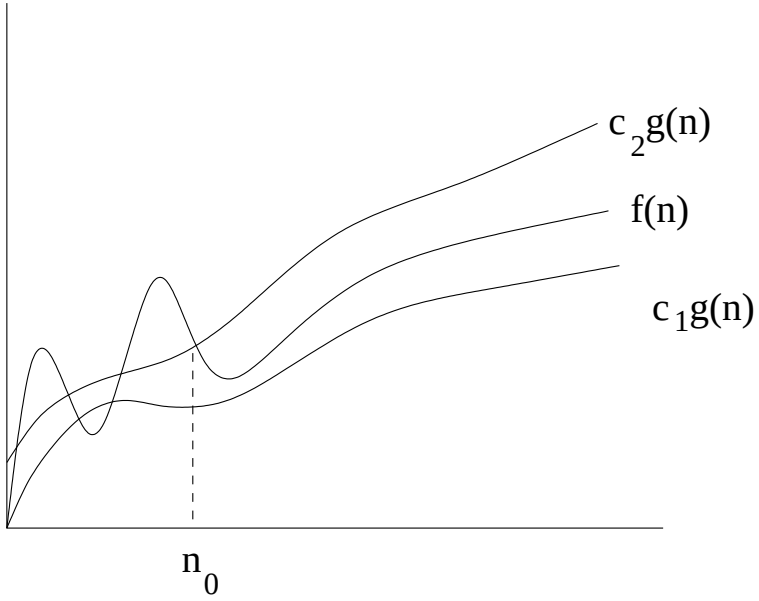
La notazione O descrive il tempo d'esecuzione di un algoritmo analizzando la sua struttura complessiva e ignorando le costanti associate al compilatore e alla macchina

L'insertion sort ha un limite superiore $O(n^2)$ nel caso peggiore

Limite asintotico stretto: notazione Theta

$\Theta(g(n)) = \{f(n)$: esistono costanti positive n_0, c_1, c_2 tali che $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ per ogni $n \geq n_0\}$.

$\Theta(g(n)) \subseteq O(g(n))$



Ogni funzione appartenente a $O(g(n))$ o a $\Theta(g(n))$ è asintoticamente non negativa.

Perché $O(g(n))$ o $\Theta(g(n))$ sia non vuoto, occorre che $g(n)$ sia asintoticamente non negativa.

Esempio Notazione Theta

Mostriamo che $n^2/2 - 3n \in \Theta(n^2)$.

Devono esistere n_0, c_1, c_2 tali che per ogni $n \geq n_0$:

$$c_1 n^2 \leq 1/2n^2 - 3n \leq c_2 n^2$$

Dividendo per n^2 (quando $n \geq 1$):

$$c_1 \leq 1/2 - 3/n \leq c_2$$

$1/2 - 3/n \leq c_2$ vale per $c_2 \geq 1/2$, e $n \geq 6$. Infatti:

$$\frac{1}{2} - \frac{3}{n} \leq c_2; \quad \text{per } n = 6: \ \frac{1}{2} - \frac{3}{6} = 0, \quad \text{per } n > 6 \ : \frac{1}{2} - \frac{3}{n} \longrightarrow \frac{1}{2}$$

$c_1 \leq 1/2 - 3/n$ vale per $c_1 \leq 1/14$, e $n \geq 7$. Infatti:

$$c_1 \leq \frac{1}{2} - \frac{3}{n}; \quad \text{per } n = 6: \ c_1 \leq 0 \text{ NO}; \quad \text{per } n = 7, \ \frac{1}{2} - \frac{3}{7} = \frac{1}{14}, \quad \text{quindi } c_1 \leq \frac{1}{14}$$

Quindi per ogni $n \geq 7$: $0 \leq 1/14 \leq 1/2 - 3/n \leq 1/2$

In generale

Se la funzione è asintoticamente positiva, i termini di grado inferiore possono essere ignorati, perché diventano trascurabili per n grande.

Prendendo come c_1 un valore leggermente più piccolo del coefficiente del termine di ordine più alto e come c_2 un valore leggermente più grande, le disuguaglianze della notazione Θ sono soddisfatte per qualche n_0.

Esempio

Mostriamo che $6n^3 \notin O(n^2)$

(Quindi anche $6n^3 \notin \Theta(n^2)$)

Supponiamo che esistano n_0, c tali che per ogni $n \geq n_0$:

$$6n^3 \leq cn^2$$

Cioè, dividendo per n^2 (per $n \geq 1$):

$$n \leq c/6$$

per un numero infinito di n.

Assurdo.

I fattori costanti (positivi) possono essere ignorati

Se $f(n)$ è $\Theta(g(n))$,
allora per ogni $d > 0$, $f(n)$ è $\Theta(d \cdot g(n))$

Se $f(n)$ è $O(g(n))$,
allora per ogni $d > 0$, $f(n)$ è $O(d \cdot g(n))$

Se $f(n)$ è $\Theta(g(n))$, allora esistono n_0, c_1, c_2, tali che per ogni $n \geq n_0$:

$$\begin{aligned} f(n) &\geq c_1 \cdot g(n) \\ &\geq c_1/d \cdot d \cdot g(n) \end{aligned} \qquad \text{e} \qquad \begin{aligned} f(n) &\leq c_2 \cdot g(n) \\ &\leq c_2/d \cdot d \cdot g(n) \end{aligned}$$

Quindi, se $c'_1 = c_1/d$ e $c'_2 = c_2/d$, per ogni $n \geq n_0$:

$$0 \leq c'_1(d \cdot g(n)) \leq f(n) \leq c'_2(d \cdot g(n))$$

Esempio

$$f(n) = 1000\,n \in \Theta(0.01\,n)$$

Sicuramente $1000\,n \in \Theta(n)$ perché $0 \leq 1000\,n \leq 1000\,n \leq 1000\,n$ ($c_1 = c_2 = 1000$).

Per il risultato precedente, se $d = 0.01$, anche $1000\,n \in \Theta(d\,n)$: se $c'_1 = c'_2 = 100\,000$

$$1 \leq 100\,000(0.01\,n) \leq 1000\,n \leq 100\,000\,(0.01\,n)$$

Regola Della Somma

Se:

$$f_1(n) \in \Theta(g(n)),$$
$$f_2(n) \in \Theta(h(n)),$$
e $g(n) \in \Theta(h(n)),$

allora

$$f_1(n) + f_2(n) \in \Theta(h(n))$$

Ipotesi:

1. $f_1(n) \in \Theta(g(n))$: esistono n_1, c_1, c_2, tali che, per ogni $n \geq n_1$: $1 \leq c_1 \cdot g(n) \leq f_1(n) \leq c_2 \cdot g(n)$
2. $f_2(n) \in \Theta(h(n))$: esistono n_2, c_3, c_4, tali che, per ogni $n \geq n_2$: $1 \leq c_3 \cdot h(n) \leq f_2(n) \leq c_4 \cdot h(n)$
3. $g(n) \in \Theta(h(n))$: esistono n_3, c_5, c_6, tali che, per ogni $n \geq n_3$: $1 \leq c_5 \cdot h(n) \leq g(n) \leq c_6 \cdot h(n)$

Se $n_0 = max(n_1, n_2, n_3)$, $c = c_1 \cdot c_5 + c_3$ e $c' = c_2 \cdot c_6 + c_4$, per ogni $n \geq n_0$:

$$\begin{aligned} f_1(n) + f_2(n) &\geq c_1 \cdot g(n) + c_3 \cdot h(n) \geq c_1 \cdot c_5 \cdot h(n) + c_3 \cdot h(n) \\ &= (c_1 \cdot c_5 + c_3)h(n) = c \cdot h(n) \end{aligned}$$

$$\begin{aligned} f_1(n) + f_2(n) &\leq c_2 \cdot g(n) + c_4 \cdot h(n) \leq c_2 \cdot c_6 \cdot h(n) + c_4 \cdot h(n) \\ &= (c_2 \cdot c_6 + c_4)h(n) = c' \cdot h(n) \end{aligned}$$

Casi Particolari della Regola della Somma

Se $f(n) \in \Theta(g(n))$,
allora
$f(n) + g(n) \in \Theta(g(n))$

La regola della somma vale anche per la notazione O

Se:

$$f_1(n) \in O(g(n)),$$
$$f_2(n) \in O(h(n)),$$
e $\quad g(n) \in O(h(n)),$

allora

$$f_1(n) + f_2(n) \in O(h(n))$$

Esempio

Dimostrare che $f(n) = 2^n + n^3$ è $O(2^n)$
(*Infatti* $n^3 \in O(2^n)$)

- Basta dimostrare che

se $n \geq 10$, $n^3 \leq 2^n$:

Per induzione su n:

- Se $n = 10$: $n^3 = 1000 < 1024 = 2^{10}$
- Ipotesi induttiva: $n^3 \leq 2^n$

$$
\begin{aligned}
(n+1)^3 &= n^3 \cdot \frac{(n+1)^3}{n^3} \\
&\leq 2^n \cdot \frac{n^3 + 3n^2 + 3n + 1}{n^3} \\
&= 2^n \cdot \left(\frac{n^3}{n^3} + \frac{3n^2 + 3n + 1}{n^3} \right) \\
&\leq 2^n \cdot \left(1 + \frac{7n^2}{n^3} \right) \\
&= 2^n \cdot \left(1 + \frac{7}{n} \right) \\
&\leq 2^n \cdot 2 \qquad\qquad (n \geq 10) \\
&= 2^{n+1}
\end{aligned}
$$

Esempi

Se $p(n)$ e $q(n)$ sono polinomi:

- Se il grado di $p(n)$ è uguale al grado di $q(n)$, allora $p(n) \in \Theta(q(n))$
- Se il grado di $p(n)$ è minore o uguale al grado di $q(n)$, allora $p(n) \in O(q(n))$
- Se il grado di $p(n)$ è maggiore del grado di $q(n)$, allora $p(n) \notin O(q(n))$
- Se $a > 1$, allora $p(n) \in O(a^n)$
- Se $a > 1$, $a^n \notin O(p(n))$

Transitività delle relazioni $f(n) \in \Theta(g(n))$ e $f(n) \in O(g(n))$

Se:

$$f(n) \in \Theta(g(n))$$
e $g(n) \in \Theta(h(n))$

Allora

$$f(n) \in \Theta(h(n))$$

Se:

$$f(n) \in O(g(n))$$
e $g(n) \in O(h(n))$

Allora

$$f(n) \in O(h(n))$$

Dimostrarlo per esercizio.

Limite asintotico inferiore: notazione Ω

$\Omega(g(n)) = \{f(n)$: esistono costanti positive n_0 e c tali che $0 \leq c \cdot g(n) \leq f(n)$ per ogni $n \geq n_0\}$

$f(n) \in \Omega(g(n))$ sse: esistono costanti positive n_0 e c tali che per ogni $n \geq n_0$:

$$0 \leq c \cdot g(n) \leq f(n)$$

$f(n) \in \Theta(f(n))$ sse $f(n) \in O(f(n))$ e $f(n) \in \Omega(f(n))$.

Quando la notazione Ω si usa per limitare inferiormente il tempo di esecuzione di un algoritmo nel caso migliore, si limita inferiormente il tempo di esecuzione su tutti gli input.

Il tempo di esecuzione dell'insertion sort per qualsiasi input è compreso tra $\Omega(n)$ e $O(n^2)$

Quando si dice $T(n) \in \Omega(g(n))$: per ogni n sufficientemente grande e per ogni input di dimensione n il tempo di esecuzione è almeno $c \cdot g(n)$ per qualche costante c.

$$f(n) = \begin{cases} n & \text{se } n \text{ è dispari} \\ n^2 & \text{se } n \text{ è pari} \end{cases}$$

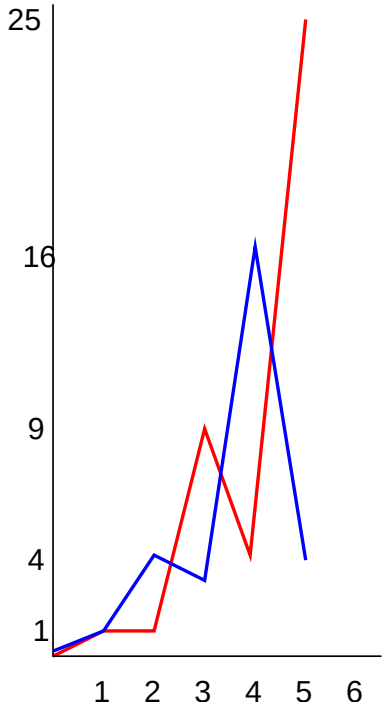
$$g(n) = \begin{cases} n^2 & \text{se } n \text{ è dispari} \\ n & \text{se } n \text{ è pari} \end{cases}$$

$$n^2 \notin O(n),$$

quindi:

$f(n)$ non è $O(g(n))$ per via degli n pari

$g(n)$ non è $O(f(n))$ per via degli n dispari



Usi impropri delle notazioni Ω e O

- Se $f(n)$ è una funzione costante, allora $f(n) \in \Theta(1)$.

- $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$

- "Operazioni" con Θ o O-grande:

  $$\Theta(n) + \Theta(n^2)$$

  "la somma di una funzione lineare e una quadratica"

  $$c_1 n + c_2 n^2$$

  per qualche $c_1, c_2 > 0$

  $$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$$

  significa che per qualche $f(n) \in \Theta(n)$, $2n^2 + 3n + 1 = 2n^2 + f(n)$.

  Regola della somma: se $f(n)$ è $O(g(n))$, allora

  $$O(f(n)) + O(g(n)) = O(g(n))$$

  Per esempio: $O(n) + O(n^2) = O(n^2)$

  "la somma di una funzione lineare e una quadratica è una funzione quadratica"

Notazione O-grande e caso peggiore

Se $T(n) \in O(g(n))$ dove $T(n)$ è il tempo di esecuzione nel caso peggiore, si limita superiormente il tempo di esecuzione su tutti gli input.

Se $T(n) \in \Theta(g(n))$ dove $T(n)$ è il tempo di esecuzione nel caso peggiore, NON si dà un limite stretto al tempo di esecuzione su tutti gli input: il tempo di esecuzione dell'insertion sort su un input già ordinato è $\Theta(n)$.

Studiando il tempo di esecuzione nel caso peggiore, spesso si determina soltanto un limite superiore.

Per determinare un limite superiore nel caso peggiore è sufficiente un'analisi più semplice

```
INSERTION-SORT (A)
1. for j <- 2 to length[A]
2.     do   key <- A[j]
4.          i <- j-1
5.          while i>0 e A[i]>key
6.                do A[i+1] <- A[i]
7.                   i <- i-1
8.          A[i+1] <- key
```

- Costo di una iterazione del ciclo while: $O(1)$ (costante).
- L'indice i può al massimo variare tra 1 ed n, quindi per j fissato il ciclo while viene eseguito al massimo n volte: per ogni j, il costo del ciclo while è $O(n)$.
- Quindi ogni iterazione nel ciclo più esterno ha costo $O(n)$.
- Il ciclo più esterno viene eseguito al massimo n volte: il costo complessivo dell'algoritmo è $O(n^2)$.

Analisi del tempo di esecuzione di un algoritmo nel caso peggiore

Istruzioni semplici

Tempo di esecuzione COSTANTE: c_i o $\Theta(1)$.

Sequenza di istruzioni semplici

Tempo di esecuzione COSTANTE: $\Theta(1)$.

Istruzioni condizionali:

```
IF <condizione> THEN <parte-then>
ELSE <parte-else>
```

Somma dei costi di:

- valutazione della condizione
- costo massimo tra il tempo di esecuzione della parte-then e della parte-else

Se il tempo di valutazione della condizione è $O(f(n))$, quello per l'esecuzione della parte-then è $O(g_1(n))$ e quello per l'esecuzione della parte-else è $O(g_2(n))$:

$$O(f(n)) + max(O(g_1(n)), O(g_2(n)))$$
$$= O(f(n)) + O(max(g_1(n), g_2(n)))$$

Cicli (for, while, repeat)

Si determina

1. un limite superiore $O(f(n))$ al numero di iterazioni nel ciclo
2. un limite superiore $O(g(n))$ al tempo per eseguire un'iterazione

Il ciclo è $O(f(n) \cdot g(n))$

```
FACT (n)
  f <- 1
  k <- n
  while k>0
    do f <- f * k
       k <- k-1
  return f
```

Analizziamo il tempo di esecuzione in termini di n.
Numero massimo di iterazioni: n Tempo di esecuzione di un'iterazione: $O(1)$
$T(n) \in O(1 \cdot n) = O(n)$

Se la misura dell'input è il numero di bit necessari per rappresentare n in binario: $k = \lceil log_2 n \rceil$:
$O(n) = O(2^k)$, quindi $T(k) \in O(2^k)$

Blocchi (sequenze di istruzioni)

Si determina un limite superiore $O(f_i(n))$ al tempo di esecuzione di ciascuna istruzione

$$O(f_1(n)) + ... + O(f_m(n))$$

Supponiamo che il programma P richiami il sottoprogramma Q

Sia $T_Q(n) \in O(f(n))$, dove n è la misura della dimensione degli input di Q.
Chiamata di Q in P: $O(f(n))$

Attenzione: si deve mettere in relazione la misura degli input di Q con quella usata da P

```
SUM-OF-FACT (m)
   sum <- 0
   n <- m
   while n>0
      do sum <- sum + FACT(n)
         n <- n-1
   return sum
```

Costo del ciclo while per n fissato: $O(1) + O(n) = O(n)$

Ciclo eseguito m volte, con $n = m, ..., 1$: costo del ciclo

$$O(m) + O(m-1) + ... + O(2) + O(1) = \sum_{j=1}^{m} O(j)$$

$$\leq \sum_{j=1}^{m} c \cdot j = c \sum_{j=1}^{m} j = c \cdot \frac{m(m+1)}{2} = O(m^2)$$

Se la misura dell'input m è il numero di bit necessari per rappresentare m in binario: $k = \lceil log_2 m \rceil$,
il costo di SUM-OF-FACT è $O(2^{2k})$.

```
FACT (n)
  if n=0 then f <- 1
         else f <- n * FACT(n-1)
  return f
```

Analizziamo il tempo di esecuzione come funzione di n:

$$T(n) = \begin{cases} a & \text{se } n = 0 \\ T(n-1) + c & \text{se } n > 0 \end{cases}$$

O, utilizzando le notazioni asintotiche:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(n-1) + \Theta(1) & \text{se } n > 0 \end{cases}$$

$T(n) = \Theta(n)$ è una soluzione di questa relazione di ricorrenza.

- Suddivisione del problema in diversi sottoproblemi (dello stesso tipo ma di dimensioni più piccole)
- Soluzione (ricorsiva) dei sottoproblemi
- Combinazione delle soluzioni dei sottoproblemi per risolvere il problema originario

A ciascun livello della ricorsione:

Divide: il problema è suddiviso in sottoproblemi

Impera: vengono risolti i sottoproblemi

Combina: le soluzioni vengono combinate

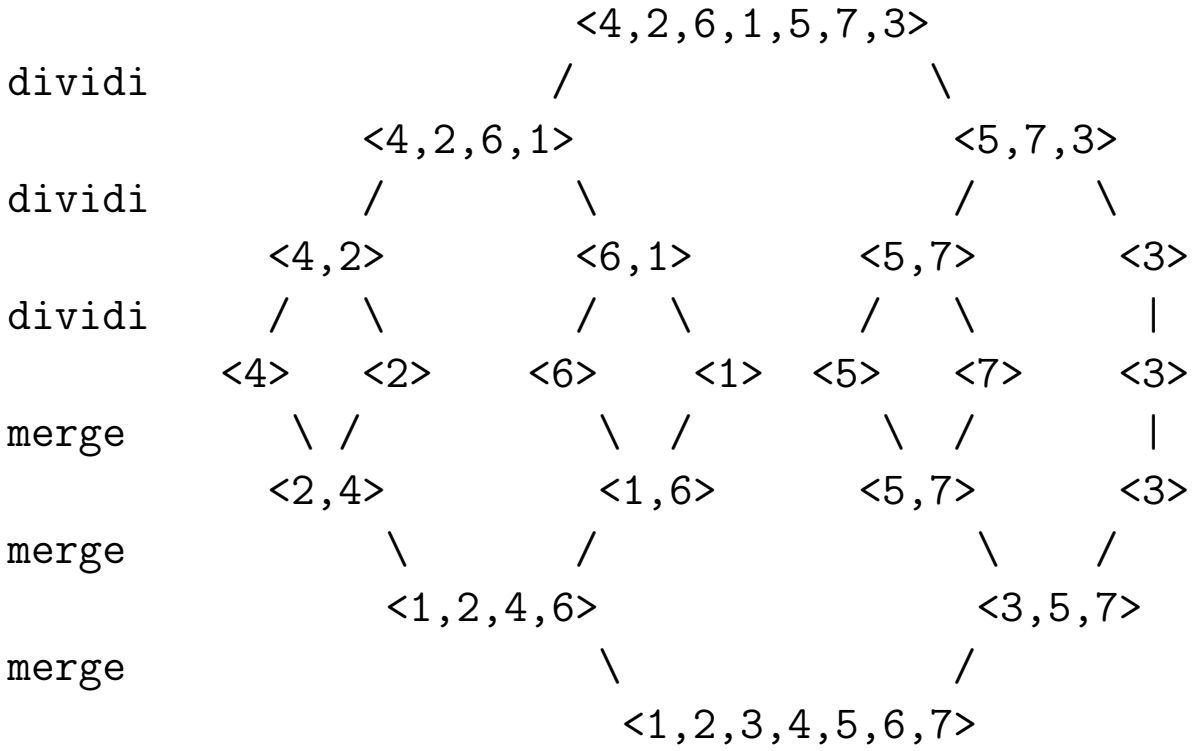
Merge sort - Ordinamento per fusione

Per ordinare la sequenza A di n elementi:

Divide: *dividere* A in due sottosequenze A_1 e A_2 di $n/2$ elementi

Impera: *ordinare*, ricorsivamente,A_1 e A_2

Combina: *fondere* le due sequenze ordinate ottenute



Merge Sort (II)

Fusione: sotto le ipotesi che:

A è un array

$p \leq q < r$ sono indici dell'array A

$A[p..q]$ e $A[q+1..r]$ sono ordinati

MERGE(A,p,q,r) fonde $A[p..q]$ e $A[q+1..r]$, generando $A[p..r]$ ordinato

MERGE-SORT(A,p,r): ordina gli elementi di $A[p..r]$

Richiamato inizialmente con MERGE-SORT(A,1,$length[A]$).

```
MERGE-SORT(A,p,r)
1.  if p< r
2.     then q <- (p+r) div 2
3.          MERGE-SORT(A,p,q)
4.          MERGE-SORT(A,q+1,r)
5.          MERGE(A,p,q,r)
```

1. Se $p \geq r$, $A[p..r]$ è vuoto oppure ha un solo elemento: il sottoarray è già ordinato.
2. `(p+r) div 2` indica la divisione intera $= \lfloor (p+r)/2 \rfloor$.

3.-4. Ordinamento ricorsivo dei due sottoarray.

5. Fusione.

Fusione

Ipotesi: $p \leq q < r$ sono indici dell'array A; $A[p..q]$ e $A[q+1..r]$ sono ordinati

```
MERGE(A,p,q,r)
i <- p      j <- q+1          k <- 1
while i<=q e j<=r
     do if A[i]<=A[j] then APP[k] <- A[i]
                           i <- i+1
                      else APP[k] <- A[j]
                           j <- j+1
        k <- k+1
if i=q+1 then for i <- j to r
                 do APP[k] <- A[i]
                    k <- k+1
         else for j <- i to q
                 do APP[k] <- A[j]
                    k<-k+1
k <- 1
for i <- p to r
   do A[i] <- APP[k]
      k <- k+1
```

Tempo di esecuzione di MERGE(A,p,q,r): se $n = r - p + 1$ è il numero di elementi in $A[p..r]$, MERGE(A,p,q,r) è $\Theta(n)$

Tempo di esecuzione di MERGE-SORT

Dimensione dell'input: numero degli elementi da ordinare ($n = length[A]$).

Assumiamo che n sia una potenza di 2, in modo che la divisione produce sempre sottoarray con $n/2$ elementi.

Costo di MERGE-SORT nel caso base: $\Theta(1)$

Divide: $D(n) = \Theta(1)$ (calcolo di $n/2$)

Impera: ogni sottoproblema ha dimensione $n/2$, i sottoproblemi sono 2: $2T(n/2)$

Combina: (MERGE) $C(n) = \Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \text{ o } n = 1 \\ 2T(n/2) + D(n) + C(n) & \text{se } n > 1 \end{cases}$$

Poiché $D(n) + C(n) = \Theta(1) + \Theta(n) = \Theta(n)$:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \text{ o } n = 1 \\ 2T(n/2) + \Theta(n) & \text{se } n > 1 \end{cases}$$

Si può risolvere questa ricorrenza con $T(n) = \Theta(n \lg_2 n)$.

Ricorrenze

Equazioni o disequazioni che descrivono una funzione in termini del suo valore su input più piccoli.

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(n-1) + \Theta(1) & \text{se } n > 0 \end{cases}$$

$$T(n) = \begin{cases} \Theta(1) & \text{se } 0 \leq n \leq 1 \\ 2T(n/2) + \Theta(n) & \text{se } n > 1 \end{cases}$$

Spesso si omette la descrizione delle "condizioni al contorno": il valore della funzione per input sufficientemente piccoli:
se $T(n)$ esprime il tempo di esecuzione di un algoritmo, normalmente $T(n) = \Theta(1)$ per n piccolo.

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

$$T(n) = 2T(n/2) + \Theta(n)$$

$$T(n) = \begin{cases} a & \text{se } n = 0 \\ T(n-1) + c & \text{se } n > 0 \end{cases}$$

$$\begin{aligned} T(0) &= a \\ T(1) &= a + c \\ T(2) &= a + 2c \\ T(3) &= a + 3c \\ &\dots \end{aligned}$$

cerchiamo di capire quale potrebbe essere la soluzione ...

... ipotizziamo:

$$T(n) = a + nc$$

e lo dimostriamo **per induzione:**

1. Passo Base: n=0 $T(0) = a = a + 0 \times c$
2. Passo Induttivo: $T(n) = a + nc$.

$$T(n+1) = T(n) + c = a + nc + c = a + (n+1)c$$

Quindi $T(n) \in \Theta(n)$.

PRIMO TEOREMA PER RISOLVERE EQUAZIONI DI RICORRENZA

Se:

$$T(n) = \begin{cases} a & \text{se } n = 0 \\ T(n-1) + g(n) & \text{se } n > 0 \end{cases}$$

allora:

$$T(n) = a + \sum_{k=1}^{n} g(k)$$

Dimostrazione

Per induzione

1. Se $n = 0$: $\sum_{k=1}^{n} g(k) = 0$, quindi $T(n) = a = a + \sum_{k=1}^{n} g(k)$.
2. Ipotesi induttiva: $T(n) = a + \sum_{k=1}^{n} g(k)$

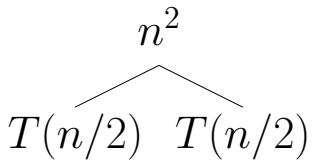
   Tesi: $T(n+1) = a + \sum_{k=1}^{n+1} g(k)$

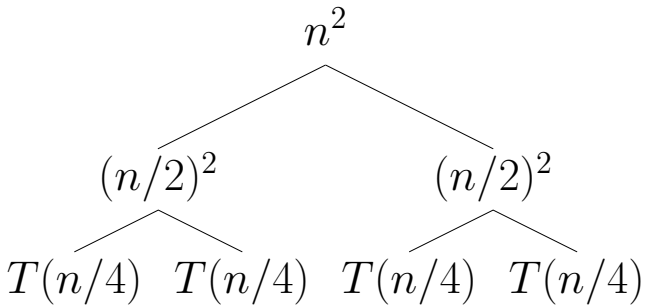
$$\begin{aligned} T(n+1) &= T(n) + g(n+1) && \text{dalla relazione di ricorrenza} \\ &= a + \sum_{k=1}^{n} g(k) + g(n+1) && \text{per ipotesi induttiva} \\ &= a + \sum_{k=1}^{n+1} g(k) \end{aligned}$$

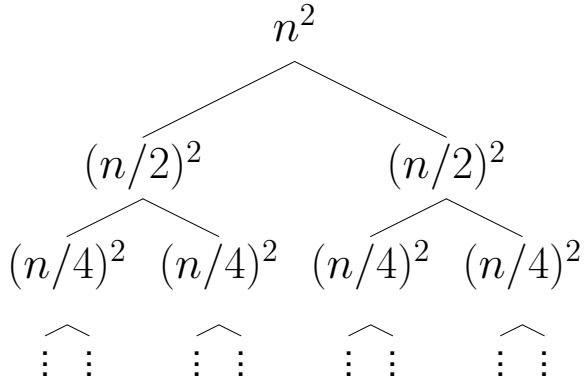
Alberi di ricorrenza

Utili soprattutto per ricorrenze che descrivono algoritmi divide-et-impera.

$$T(n) = 2T(n/2) + n^2$$







$\longrightarrow \quad n^2 = \frac{1}{2^0} n^2$

$\longrightarrow \quad \frac{1}{2} n^2 = \frac{1}{2^1} n^2$

$\longrightarrow \quad \frac{1}{4} n^2 = \frac{1}{2^2} n^2$

Totale: $\Theta(n^2)$

Le foglie sono $(\frac{n}{2^k})^2 = \Theta(1)$

Infatti se $n = 2^k$, l'albero ha altezza $k = lg\ n$ si ha:

$$n^2 \frac{1}{2^0} + n^2 \frac{1}{2^1} + n^2 \frac{1}{2^2} + \cdots + n^2 \frac{1}{2^k} = n^2 \sum_{i=0}^{k} \frac{1}{2^i}$$

Poiché se $q < 1, \quad \Sigma_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}$

$$n^2 \sum_{i=0}^{k} \frac{1}{2^i} = n^2 \sum_{i=0}^{k} \left(\frac{1}{2}\right)^i = n^2 \frac{(\frac{1}{2})^{k+1} - 1}{\frac{1}{2} - 1} = n^2(-2)(\frac{1}{2^{k+1}} - 1) = 2n^2(1 - \frac{1}{2^{k+1}}) =$$

$$2n^2(\frac{2^{k+1} - 1}{2^{k+1}}) = n^2 \frac{2 \cdot 2^k - 1}{2^k} = (n = 2^k) = n^2 \frac{2n - 1}{n} = n(2n - 1) = 2n^2 - n < 2n^2$$

Master Theorem per la Soluzione di Equazioni di Ricorrenza (Forma Semplice)

Sia $p(n^k)$ un polinomio di grado k.
Se

$$T(n) = \begin{cases} c_0 & \text{se } n = 0 \\ a\,T(n/b) + p(n^k) & \text{se } n > 0 \end{cases}$$

per $a, b \geq 1$ e $p(n^k)$ un polinomio di grado k.

Allora:

caso 1: $T(n) = \Theta(n^{log_b a})$ se $a > b^k$.
caso 2: $T(n) = \Theta(n^k lg\, n)$ se $a = b^k$.
caso 3: $T(n) = \Theta(n^k)$ se $a < b^k$.

Esempi

Ricorrenza del MERGE-SORT:

$$T(n) = 2T(n/2) + \Theta(n) = 2T(n/2) + c \cdot n$$

$a = 2,\ b = 2; \qquad p(n^k) = c \cdot n, \quad k = 1$

Quindi $a = b^k$, e si applica il caso 2:

$$T(n) = \Theta(n^k \, lg \, n) = \Theta(n \cdot lg \, n)$$

$$T(n) = 9T(n/3) + n$$

$a = 9,\ b = 3; \qquad p(n^k) = n, \quad k = 1$

Quindi $a > b^k$ e si applica il caso 1:

$$T(n) = \Theta(n^{log_b a}) = \Theta(n^{log_3 9}) = \Theta(n^2)$$

$$T(n) = T(2n/3) + 1$$

$a = 1,\ b = 3/2; \qquad p(n^k) = 1, \quad k = 0$

Quindi $a = b^k$ e si applica il caso 2:

$$T(n) = \Theta(n^k \, lg \, n) = \Theta(lg \, n)$$