La notazioni asintotica

Introduciamo la prima delle notazioni utilizzate per esprimere la complessità di tempo asintotica di un algoritmo: la notazione Θ

(Theta grande, nel nostro contesto solo theta).

Vedremo alcuni esempi per chiarirne la definizione, altri per chiarirne l'uso nell'analisi degli algoritmi.

Introduciamo poi anche le altre notazioni asintotiche utilizzate per esprimere la complessità di tempo asintotica di un algoritmo: la notazione "O" grande e Ω grande.

Per gli argomenti trattati in questi lucidi si veda il capitolo 3 del testo consigliato [CLRS].

Ignoriamo le costanti!

<u>Obiettivo</u>: semplificare l'analisi eliminando l'informazione non necessaria (come arrotondare 1.000.001≈1.000.000)

Introduciamo uno strumento formale per dire che $3n^2 \approx n^2$ o che $4n+2 \approx n$:

la notazione "Θ" (Teta)

Definizione algebrica:

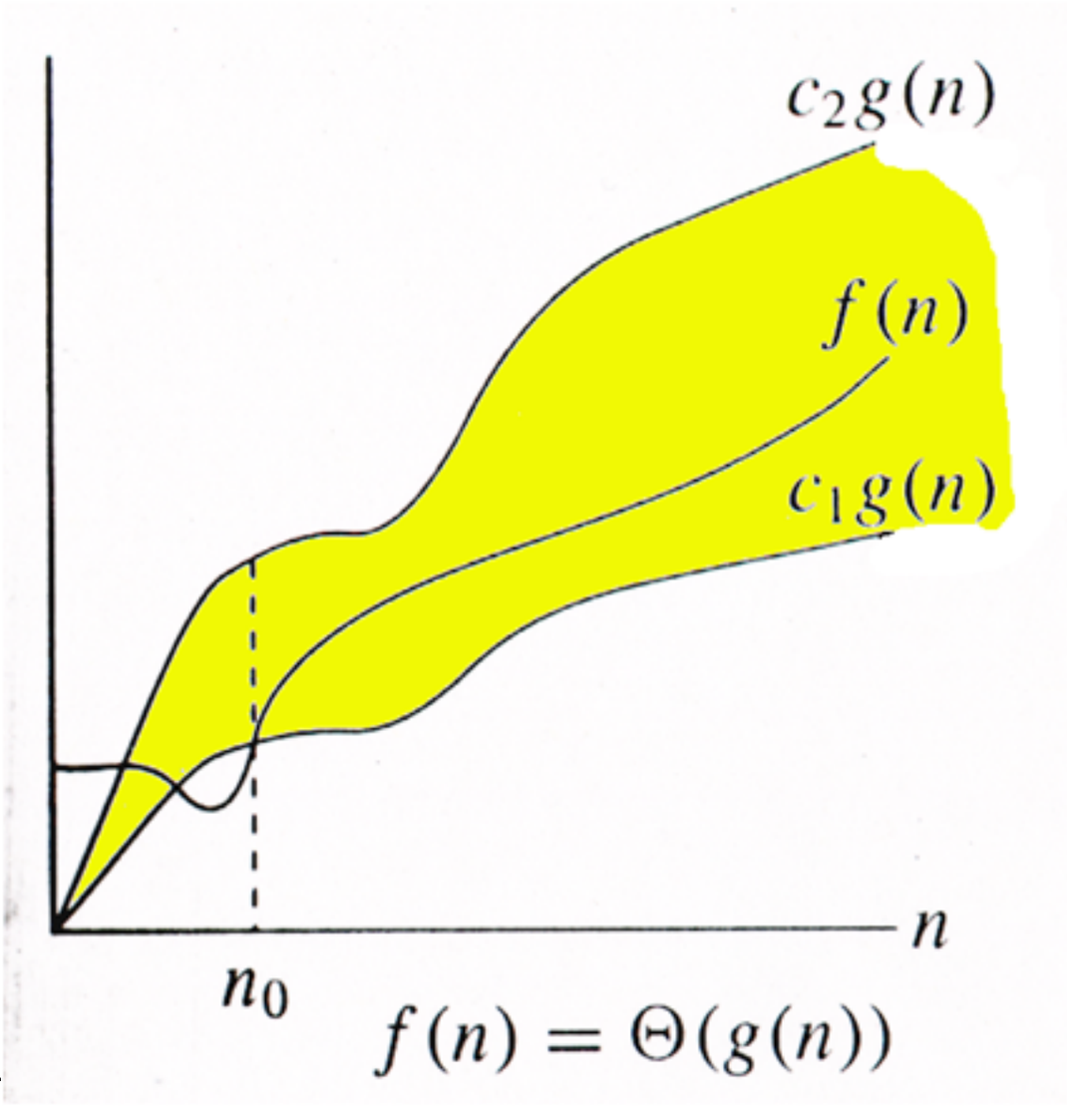
Date due funzioni, *f(n)* e *g(n)*, che possiamo pensare da N a N, o da N a R (i reali) a seconda della convenienza, diciamo che

$$f(n) \text{ è } \Theta(g(n))$$

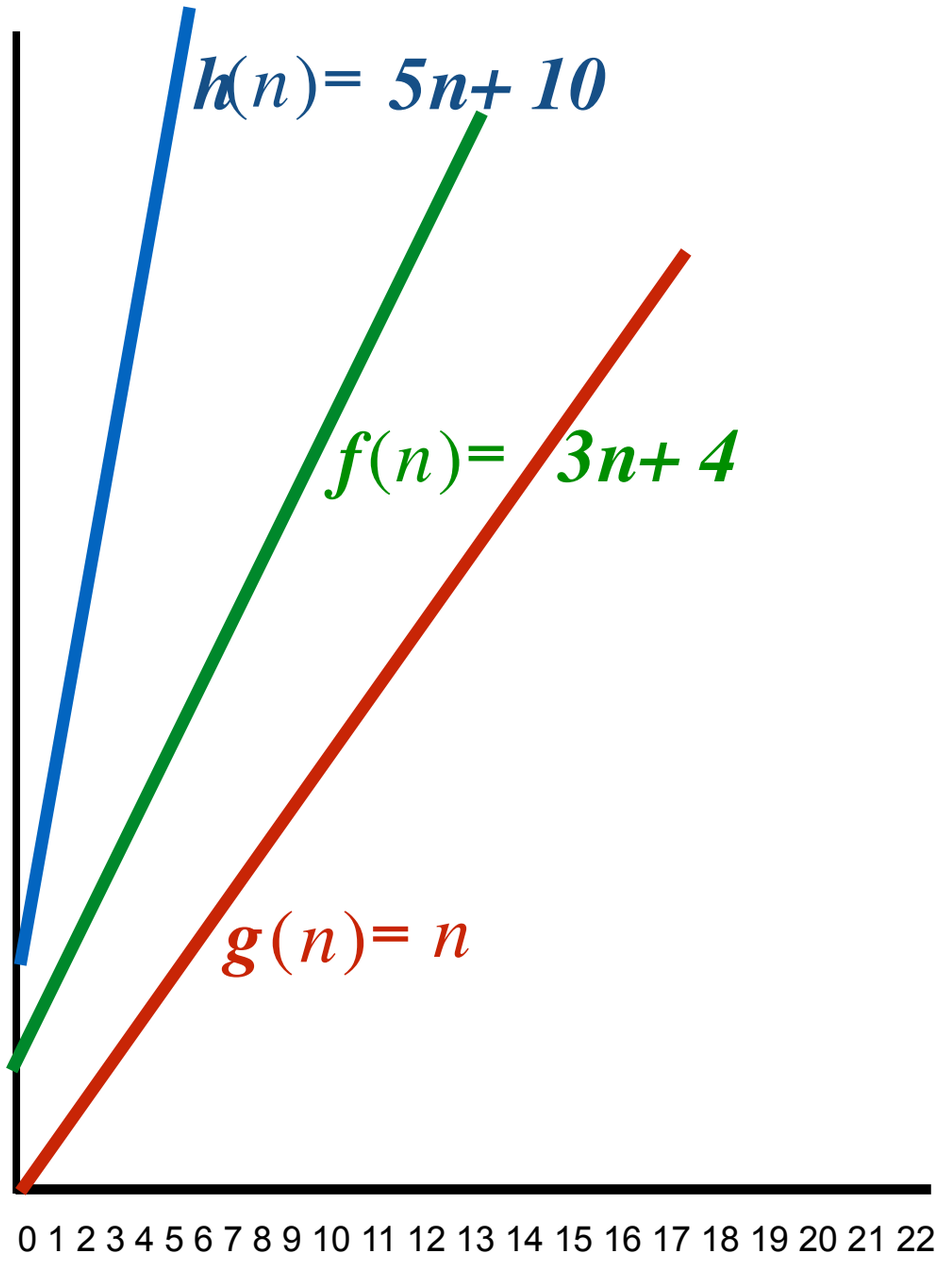
se ci sono tre costanti positive c_1 , c_2 e n_0 tali che

$$0 \le c_1\, g(n) \le f(n) \le c_2\, g(n)$$

per ogni $n \ge n_0$



Per esempio tutte le funzioni lineari sono in $\Theta(n)$



Possiamo dimostrare che h(n) = 5n+10 è $\Theta(n)$, così come f(n) = 3n+4 è $\Theta(n)$.
Infatti la funzione identità g(n)=n è la più semplice funzione lineare.
Il tasso di crescita non è influenzato dal valore delle costanti.

Tasso di crescita e Θ

La notazione Θ ci consente di esprimere succintamente il tasso di crescita di una funzione

f(n) è Θ(g(n)) significa che il tasso di crescita di f(n) è uguale a quello di g(n)

La definizione esatta è

Θ(g(n)) = {f(n) | $\exists$ tre costanti positive c_1 , c_2 e n_0 tali che 0 ≤ c_1 $g(n)$ ≤ $f(n)$ ≤ c_2 $g(n)$ per $n \geq n_0$}

Si dovrebbe dire $f(n) \in \Theta(g(n))$, ma si usa dire

$f(n)$ è $\Theta(g(n))$ e scrivere $f(n) = \Theta(g(n))$

compiendo un abuso di notazione.

Le principali funzioni

f(n) = 10.000 ogni funzione costante è in $\Theta(1)$

f(n) = log n (logaritmo in qualunque base) logaritmica

f(n) = n lineare

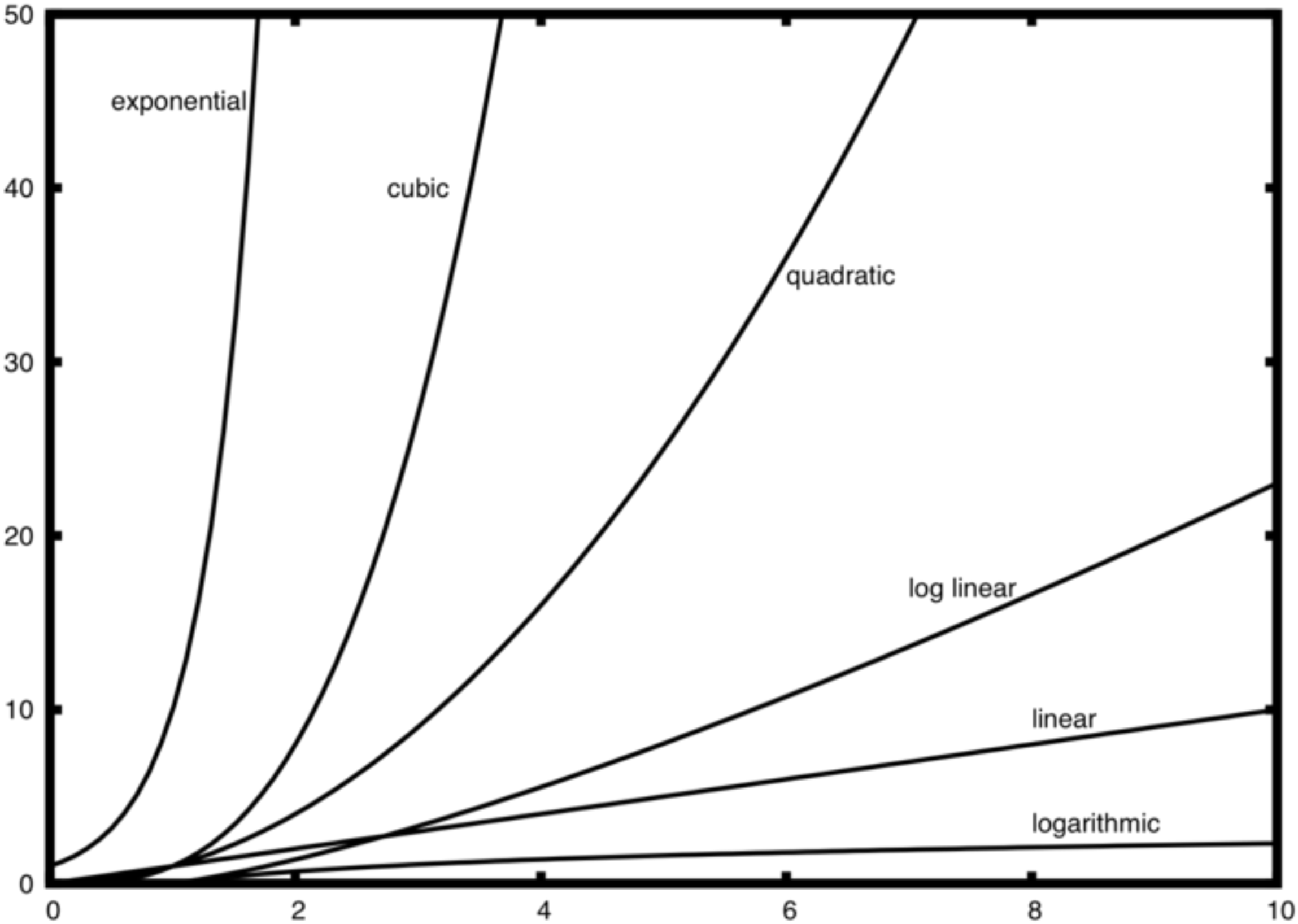
f(n) = nlog n log linear

f(n) = n^2 quadratica

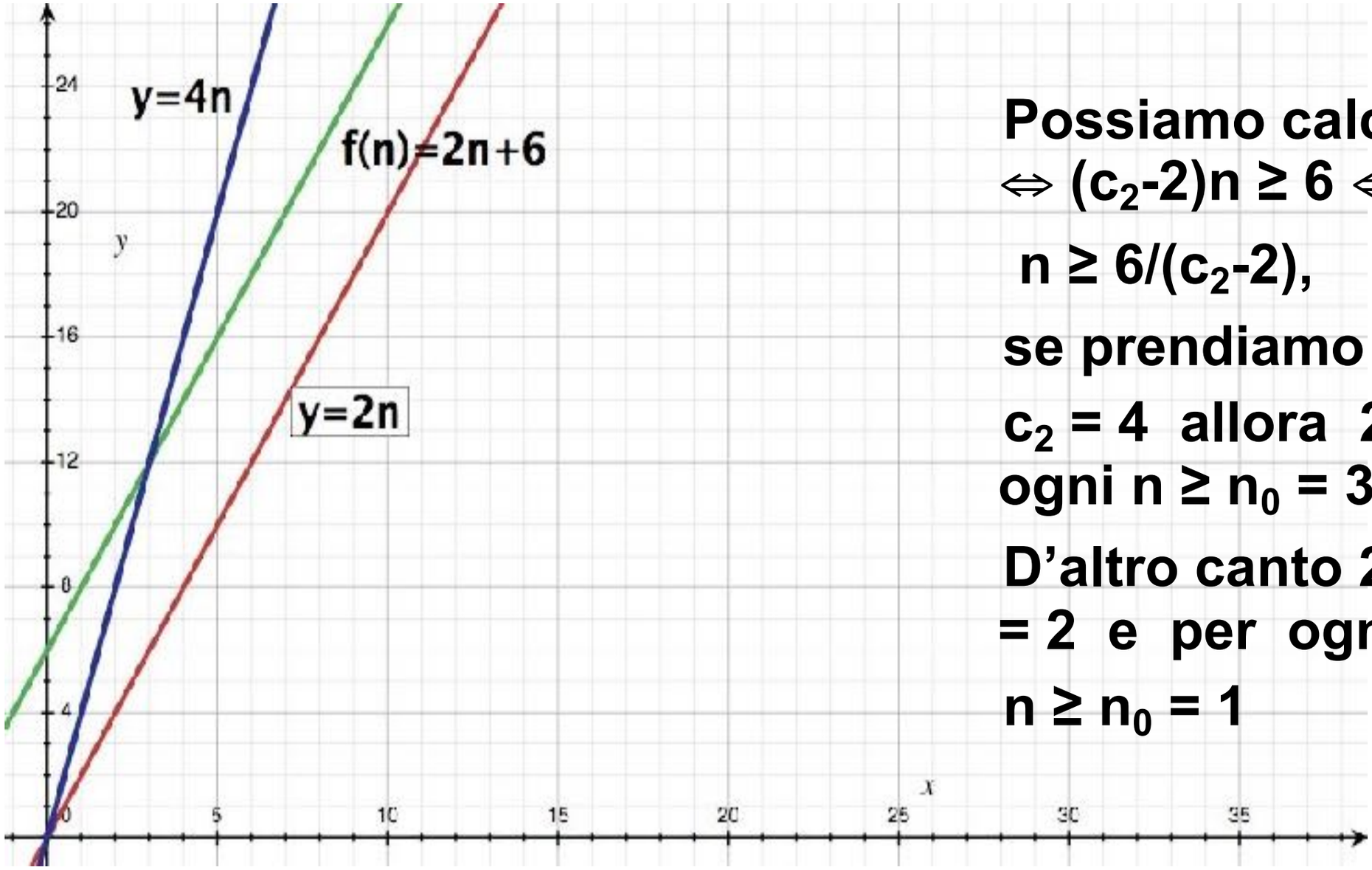
f(n) = n^3 cubica

f(n) = 2^n esponenziale

Crescita principali funzioni



Esempio 1



Possiamo calcolare: $2n+6 \leq c_2 n$ $\Leftrightarrow (c_2-2)n \geq 6 \Leftrightarrow$

$n \geq 6/(c_2-2)$,

se prendiamo

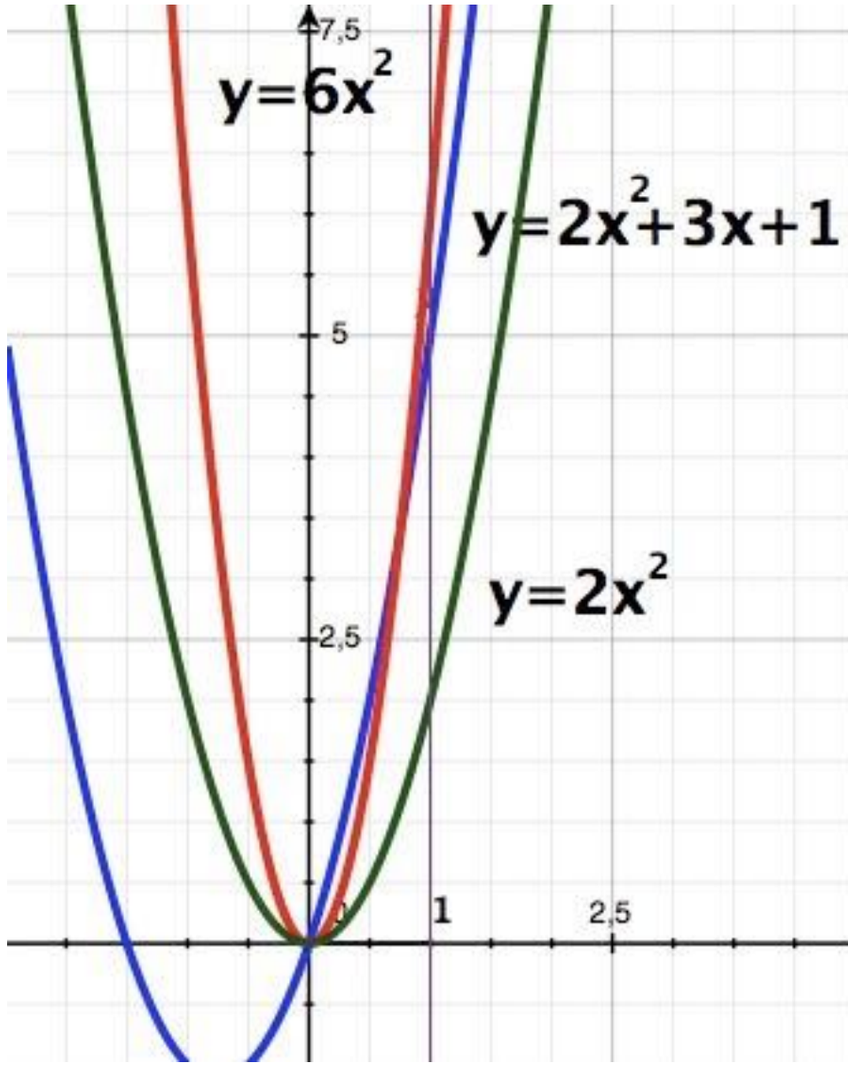
$c_2 = 4$ allora $2n+6 \leq 4n$ per ogni $n \geq n_0 = 3$.

D'altro canto $2n+6 \geq c_1 n$ per $c_1 = 2$ e per ogni $n \geq n_0 = 1$

Facciamo vedere che $f(n) = 2n+6 = \Theta(n)$.

Posto $g(n) = n$ bisogna trovare tre costanti c_1, c_2 e n_0 tali che $c_1 g(n) \leq f(n) \leq c_2 g(n)$ per $n \geq n_0$

Esempio 2



Il tasso di crescita non è influenzato dal valore delle costanti e nemmeno dai termini di ordine inferiore.

$f(x) = 2x^2+3x+1 = \Theta(x^2)$, infatti

$2x^2+3x+1 \leq 6x^2$, per $x \geq 1$,

e $2x^2+3x+1 \geq 2x^2$ per $x \geq 0$

Esempio 3

$f(x) = x^3+2x^2+1 = \Theta(x^3)$,

infatti $x^3+2x^2+1 \leq 4x^3$ e $x^3+2x^2+1 \geq x^3$ per $x \geq 1$,

Polinomi

Dato un polinomio

$$p(n)= a_d n^d + a_{d-1} n^{d-1} +...+ a_1 n + c,$$

si può dimostrare che se $a_d > 0$ allora $p(n) = \Theta(n^d)$

Si può dimostrare osservando che

$p(n) \leq (a_d + |a_{d-1}|+...+|a_1|+ |c|)n^d$ per ogni $n \geq 1$, per cui

$c_2 = (a_d + |a_{d-1}|+...+|a_1|+ |c|)$

e che $p(n) \geq 1/2 a_d n^d$, per n abbastanza grande, prendendo quindi

$c_1 = 1/2 a_d$

Il termine di grado massimo è <u>dominante</u>, perchè è quello che cresce più in fretta.

Polinomi

Vogliamo far vedere che $p(n) \geq 1/2 a_d n^d$ per n abbastanza grande:

$p(n) = a_d n^d + a_{d-1} n^{d-1} + \ldots + a_1 n + c =$

$1/2 a_d n^d + 1/2 a_d n^d + a_{d-1} n^{d-1} + \ldots + a_1 n + c$
poiché $a_d n^d = 1/2 a_d n^d + 1/2 a_d n^d$

e poiché $1/2 a_d n^d = 1/(2d) a_d n^d + 1/(2d) a_d n^d + \ldots + 1/(2d) a_d n^d$

$1/2 a_d n^d + 1/2 a_d n^d + a_{d-1} n^{d-1} + \ldots + a_1 n + c =$
$1/2 a_d n^d + (1/(2d) a_d n^d + a_{d-1} n^{d-1}) + (1/(2d) a_d n^d + a_{d-2} n^{d-2}) \ldots + (1/(2d) a_d n^d + c)$
ma visto che $a_d > 0$ allora, per $n \geq 2d(|a_d| + |a_{d-1}| + \ldots + |a_1| + c)$
$(1/(2d) a_d n^d + a_{d-1} n^{d-1}) \geq 0$
…
$(1/(2d) a_d n^d + c) \geq 0$

Quindi $1/2 a_d n^d \leq p(n) \leq (a_d + |a_{d-1}| + \ldots + |a_1| + |d|) n^d$ per $n \geq 2d(|a_d| + |a_{d-1}| + \ldots + |a_1| + c)$

dettaglio di una delle disuguaglianze

Visto che $a_d > 0$ allora, per $n \geq 2d|a_d| + 2d|a_{d-1}| + \ldots + 2d|a_1| + 2dc$

si ha $(1/(2d)a_d n^d + a_{d-1}n^{d-1}) \geq 0$ infatti

$(1/(2d)a_d n^d + a_{d-1}n^{d-1}) = n^{d-1}(1/(2d)a_d n + a_{d-1})$ e

$n^{d-1}(1/(2d)a_d n + a_{d-1}) \geq 0$ è vero se

$n^{d-1} \geq 0$ e $(1/(2d)a_d n + a_{d-1}) \geq 0$,

$(1/(2d)a_d n + a_{d-1}) \geq 0 \Leftrightarrow (1/(2d)a_d n \geq - a_{d-1}) \Leftrightarrow (a_d n \geq - 2da_{d-1}) \Leftrightarrow$

$n \geq - 2da_{d-1}/a_d$

Se $a_{d-1} < 0$ prendendo $n \geq 2d|a_d| + 2d|a_{d-1}| + \ldots + 2d|a_1| + 2dc$
la disuguaglianza è verificata perchè
$2d|a_d| + 2d|a_{d-1}| + \ldots + 2d|a_1| + 2dc \geq -2da_{d-1}/a_d$

Polinomi

Qualche volta serve mettere in evidenza alcuni aspetti della complessità di un algoritmo e allora non si usa completamente il potere sintetico della notazione.
Per esempio si può anche scrivere espressioni del tipo

$$f(n) = \Theta(n^3) + \Theta(n),$$

per sottolineare che la crescita è cubica, con un termine additivo lineare.

Il termine dominante qui è $\Theta(n^3)$, infatti $f(n) = \Theta(n^3)$.

logaritmi e funzioni lineari

Data la funzione

$$f(n) = an + b\lg n + d, \text{ con } a,b,d > 0$$

si può dimostrare che $f(n) = \Theta(n)$.

Infatti $\lg n \leq n$, per ogni $n > 0$ quindi

$f(n) = an + b\lg n + d \leq 2\max\{a,b\}\, n + d \leq 3\max\{a,b,d\}n$, per $n > 0$

Inoltre $f(n) = an + b\lg n + d \geq a\, n + d \geq an$, per $n > 0$.

Quindi $f(n) = \Theta(n)$, perchè esistono tre costanti positive
$c_1 = a$,
$c_2 = 3\max\{a,b,d\}$ e
$n_0 = 1$ tali che

$$0 \leq c_1\, n \leq f(n) \leq c_2\, n \text{ per } n \geq n_0$$

logaritmi e loro base

Si può dimostrare che $\log_a n = \Theta(\lg n)$, per ogni possibile base $a > 0$ e $a \neq 1$.

In generale vale che $\log_a n = \log_c n / \log_c a$ ($c>0$ e $c\neq1$)

Quindi $\log_a n = \lg n / \lg a$.

Per definizione di Θ devono esistere c,d e n_0 tali che $c \lg n \leq \log_a n \leq d \lg n$ per ogni $n \geq n_0$

Basta prendere $c = d = 1/\lg a$ e $n_0 = 1$

Analisi asintotica degli algoritmi

- **Con l'analisi asintotica degli algoritmi determiniamo il tasso di crescita di una funzione che esprime il tempo di calcolo di un algoritmo.**

- **Come procediamo con la notazione Θ:**
  - **Calcoliamo il massimo numero di esecuzioni delle operazioni per una certa dimensione n dell' input (caso peggiore!)**
  - **Esprimiamo questa funzione con la notazione Θ.**

Perché il caso peggiore?

- **Perché dà un limite superiore per ogni input**
- **In alcune applicazioni può essere un caso piuttosto frequente.**
  - **Per esempio nella ricerca sequenziale ogni volta che un elemento non è presente.**
- **Il caso medio, che presuppone la conoscenza della distribuzione di frequenza dell'input, può essere**
  - **molto difficile da calcolare o**
  - **non discostarsi molto dal caso peggiore o**
  - **dare risultati irrealistici perché le ipotesi sulla distribuzione dell'input sono semplicistiche**

Non affronteremo il calcolo del caso medio

L'esempio dell'inserimento in una lista ordinata

Nel caso dell'inserimento di un elemento in una lista ordinata di interi, prima soluzione, avevamo ottenuto che nel caso peggiore il tempo era espresso da una funzione lineare nel numero degli elementi

$$TMAX_{InsOrd1}(n) = c * n + d,$$

quindi possiamo più brevemente dire che l'algoritmo ha, nel caso peggiore, una complessità lineare, cioè in Θ(n).

Con la seconda soluzione eravamo arrivati ad una soluzione del tipo

$TMAX_{InsOrd2}(n) = TMAX_{RBisect}(n) + TMAX_{insPos}(n) = c_i * n + d_i + c_r * \lg n + d_r$ **che si può semplificare in prima battuta scrivendo**

$$\Theta(n) + \Theta(\lg n)$$

che si può ulteriormente semplificare con Θ(n), come abbiamo visto.

Conclusione inserimento in una lista ordinata

Quindi asintoticamente le due soluzioni sono equivalenti. Si può dire che un andamento lineare è caratteristico dell'algoritmo di inserimento di un elemento in una lista ordinata.

Per capire quale metodo conviene di più bisogna calarsi in un contesto specifico, relativo alla dimensione della lista e dei numeri su cui si devono fare confronti.

L’esempio dell’inserimento in una lista ordinata

Se supponiamo che

$$TMAX_{InsOrd1}(n) = (100 + 2)n + d,$$

e che

$$TMAX_{InsOrd2}(n) = 2n + 100 \lg n + d$$

allora certamente la seconda soluzione è migliore della prima infatti

$$(100 + 2)\ n + d \geq 2 * n + 100 * \lg n + d$$

$$100\ n \geq 100 \lg n \quad \text{è vero per ogni } n \geq 1$$

Esercizio di calcolo complessità: la ricerca sequenziale

INPUT: una sequenza A di n numeri e l'elemento da cercare, *item*.
OUTPUT: se esiste i, $0 \le i \le len(A) - 1$ tale che $A[i] = item$ e $A[k] \neq item$, per k=0,..i-1, allora il valore restituito è i, -1 se item non è presente.

```
SeqS(A, item)
pos = 0                                          Θ(1)
n = len(A)
finchè pos < n and A[pos] ≠ item                 tra Θ(1) e Θ(n)
      pos = pos +1
if pos = n then return -1 else return pos        Θ(1)
```

Quindi SeqS (Sequential Search) ha un tempo di calcolo nel caso peggiore in Θ(n) e nel caso migliore in Θ(1).
Notiamo che in ogni ciclo il confronto necessario per stabilire se eseguire ancora le istruzioni all'interno del ciclo è eseguito una volta di più rispetto all'esecuzione di queste ultime. Per esempio nel caso di SeqS, quando l'elemento non è presente i confronti tra A[pos] e item sono n, ma pos è confrontato ancora una volta con n.

Esercizio di calcolo complessità: la ricerca binaria

BinS(A,k)

INPUT: una sequenza A di elementi e l'elemento da cercare, k.

PREC: A[0]≤A[1]≤...≤A[*n*-1], dove *n* è il numero degli elementi

OUTPUT: se k è presente, restituisce la posizione di k in A, altrimenti -1

```
lo = 0 e hi = len(A)
%nel ciclo che segue l'intervallo della ricerca è sempre determinato
    dagli indici lo,…,hi-1
while  lo < hi:
     m = lo + (hi-lo)/2
    se k = A[m] return m
    se k < A[m] poni hi=m
%cerca k tra gli elementi di indice lo,…,m-1,
    se k > A[m]   poni lo=m+1
%cerca k tra gli elementi di indice m+1,…,hi-1
return -1
```

Quindi BinS (Binary Search) ha un tempo di calcolo nel caso peggiore in Θ(lg n), come per RBISECT, ma nel caso migliore in Θ(1).

calcolo del minimo in un array

```
Min(A)
input: A è una lista di interi
Output: l'indice del minimo in A
corrMin = 0
n = len[A]
i = 1
while (i < n):
   if (A[i] < A[corrMin]) then  corrMin = i
   i = i+1
return corrMin
```

Θ(1)

Θ(1)

Poichè il ciclo viene eseguito n volte e le operazioni di confronto e assegnamento coinvolte nel ciclo vengono eseguite in tempo costante si ha, in tutti i casi
$T(n) = \Theta(n) + \Theta(1) = \Theta(n)$

Frammenti di codice

```
for ( int i = 1;i ≤ n;i++)  {
      for (int j = 1;j ≤ i;j++) {
        print(“*");
       }
}
```

Θ(1)

Θ(i)

?

Il ciclo for, annidato nel primo, ha una complessità di tempo pari a Θ(i), con i=1,...,n
quindi il ciclo for esterno ha complessità pari alla somma per i=1 fino a n di Θ(i), cioè $\Theta(n(n+1)/2) = \Theta(n^2/2) = \Theta(n^2)$

Esercizio 2

```
elMatrice (A) {
n ← numRighe[A]
m ← numColonne[A]
for ( int j = 1;j ≤ m;j++) {
      for ( int i = 1;i ≤ n;i++) {
            A[1,j] = A[1,j] + A[i,j]
          }
     if  A[1,j] ≠0 then
          for ( int k = 1;k ≤ j;k++) {
               A[1,j] ← A[1,j] + A[1,k]
          }
}}
```

Se supponiamo che $n = \Theta(m)$, allora $TMAX(n) = \Theta(m^2)$

$\Theta(1)$

$\Theta(m(n + m))$

$\Theta(n)$

$\Theta(1)$

tra $\Theta(1)$ e $\Theta(j)$ per j=1,...,m

$\Theta(1)$

Nel caso peggiore il terzo for, annidato nel primo, ha una complessità di tempo pari a $\Theta(j)$, per j=1,...,m, quindi concludiamo che questa parte del ciclo ha una complessità $\Theta(m(m+1)/2) = \Theta(m^2)$. Aggiungendo a questo la complessità del primo ciclo for annidato, otteniamo $\Theta((n*m+m^2)$

Alla stessa conclusione, ma non sempre questo è vero, arriviamo maggiorando il costo di ogni esecuzione del terzo ciclo con $\Theta(m)$, per cui $\Theta(m)+ \Theta(n)$, m volte dà $\Theta(n*m + m^2)$ e di nuovo otteniamo $TMAX(n)= \Theta(n*m + m^2)$.

Esercizio 3

```
1. i=1                                          Θ(1)
   while (i < n) {
        A[i] = A[i]*B[i]                        Θ(1)
        i = i+2                                                  Θ(lg n) + Θ(1) =
        j = 1                                                    Θ(lg n)
        while (j<n) {                           Θ(lg n)
                A[j] = B[i]
                j=2*j
        }
   }
```

Il ciclo while interno viene eseguito un numero di volte pari al numero di raddoppi necessari, partendo da 1, per eguagliare o superare n. Se $2^k \leq n < 2^{k+1}$ saranno necessarie al più k + 1 raddoppi. Poichè le istruzioni all'interno del ciclo sono eseguite in tempo costante, il ciclo interno ha complessità Θ(lg n).

Quante volte viene eseguito il ciclo esterno?
circa n/2 volte
Quindi $TMAX_{Es3}(n) = \Theta(1) + n/2 * \Theta(\lg n) = \Theta(n/2 \lg n) = \Theta(n \lg n)$

Più preciso, meno sintetico

Più sintetico

Analisi di frammenti di codice

```
c = 1
m = n*n
while m>1 do
    for j=1 to m do c++
    m = m/2
```

Il ciclo for interno, le cui istruzioni interne sono di costo costante, viene eseguito la prima volta m volte, poi m/2 volte, m/4,…, fino a quando m≤1. Se k è tale che $2^k \le m < 2^{k+1}$ vuol dire che dobbiamo valutare la somma

$m + m/2 + m/4,\ldots,m/2^k =$

$$\sum_{i=0,\ldots,k} m(1/2)^i = m \sum_{i\in[0,k]} (1/2)^i < m \sum_{i\in[0,\infty)} (1/2)^i < m\,\Theta(1).$$

Quindi $m\sum_{i\in[0,k]}(1/2)^i < m\,\Theta(1)$, inoltre m è anche un limite inferiore al numero di esecuzioni del ciclo.

Infine m = n*n e concludiamo che il frammento di codice viene eseguito in $\Theta(n^2)$.

Analisi di frammenti di codice

Se FUN(m) richiede tempo $\Theta(m^2)$, qual è il tempo di esecuzione di questo ciclo?

```
for i=1 to n do
    j=n                          Θ(1)
    while j>1 do                          Θ(n²)
      FUN(j)          Θ(j²)
      j=j/2
```

Il ciclo while interno viene eseguito un numero di volte pari al numero di divisioni necessarie, partendo da n, per arrivare a 1.
Se $2^k \le n < 2^{k+1}$ saranno necessarie k esecuzioni del ciclo while.
Le istruzioni all'interno del ciclo sono eseguite in tempo $\Theta(j^2)$, per j=n,n/2,…,1
Il tempo totale di esecuzione del ciclo while è
$\Theta(n^2) + \Theta((n/2)^2) + \ldots + \Theta((n/2^k)^2) =$

$$\sum_{i=0,\ldots,k} \Theta((n/2^i)^2) = \sum_{i=0,\ldots,k} \Theta(n^2)(1/2^i)^2 = \Theta(n^2) \sum_{i=0,\ldots,k} (1/2^2)^i =$$

$$\Theta(n^2) \sum_{i=0,\ldots,k} (1/4)^i$$

Analisi di frammenti di codice

Se FUN(m) richiede tempo $\Theta(m^2)$, qual è il tempo di esecuzione di questo ciclo?

```
for i=1 to n do
    j=n
    while j>1 do
      FUN(j)
      j=j/2
```

$\Theta(1)$

$\Theta(n^2)$

$\Theta(n^3)$

$\Theta(j^2)$

Si deve valutare la somma dei quadrati di $1/4^i$, per i=0 fino a k, cioè la somma delle potenze successive di 1/4. Questa somma può essere limitata superiormente dalla sua estensione all'infinito. La serie di potenze di ragione 1/4 converge a una costante.
Poiché n^2 è anche un limite inferiore al tempo di esecuzione del ciclo interno, otteniamo che il ciclo interno ha una complessità $\Theta(n^2)$. L'intero frammento ha quindi complessità $\Theta(n^3)$ e non limitato superiormente da $cn^3 \lg n$, come avremmo potuto concludere con la maggiorazione di j con n.

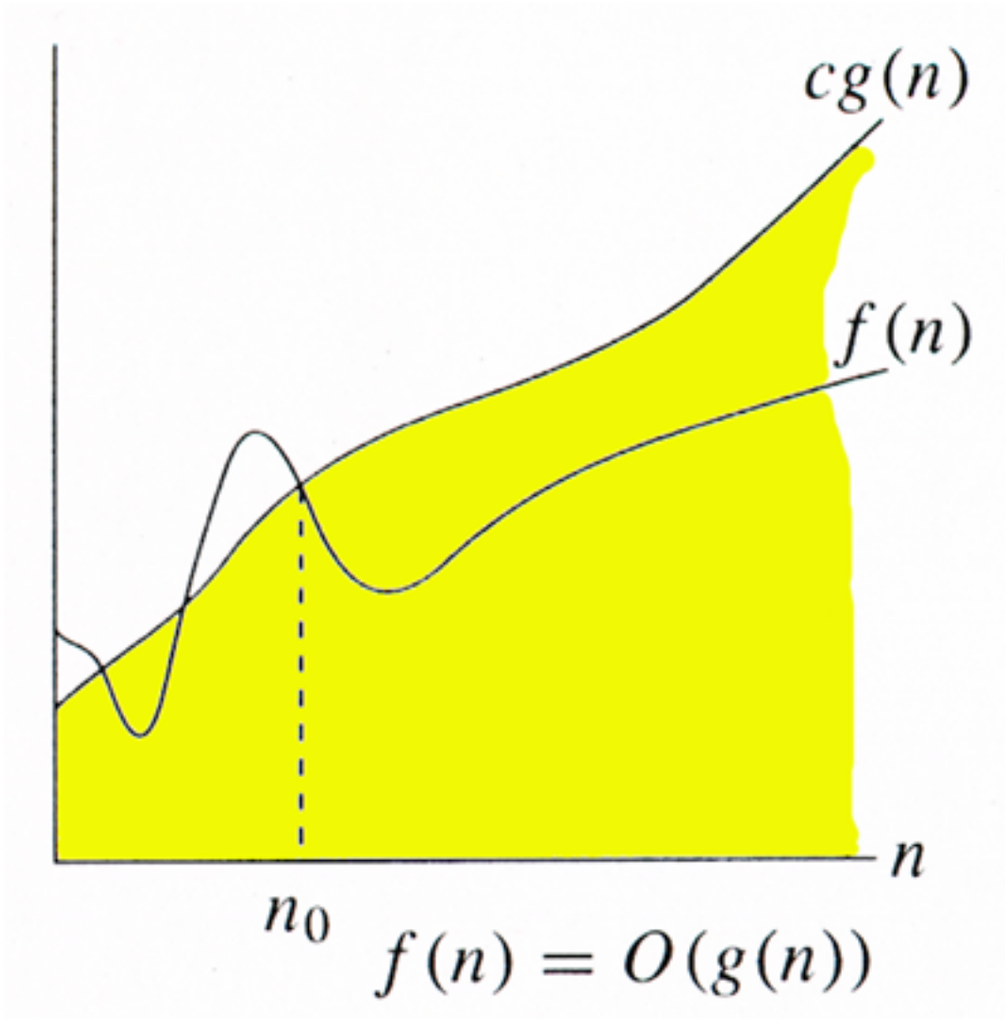
O grande

Analogamente a Θ, O grande consente di eliminare le costanti, ma questa volta per esprimere un limite superiore alla crescita della funzione

O(g(n)) = {f(n) | ∃ due costanti positive c e n_0 e
$0 \le f(n) \le c\, g(n)$ per $n \ge n_0$ }

Se f(n) = O(g(n)) allora

g(n) è un limite superiore asintotico per f(n)



O grande: esempi

$n^2 = O(n^3)$ Nota che $n^2 \neq \Theta(n^3)$

$n^2 \neq O(n)$

Funzioni in $O(n^2)$:

n $n/1000$ $n^{1.999}$ n^2 n^2+n $1000n^2+500000n$

Funzioni non in $O(n^2)$:

n^3

$n^{2.5}$

2^n

O grande: esempio di prova algebrica

Dimostriamo che $2n^2 = O(n^3)$.

Per definizione di O bisogna che esistano $n_0, c \geq 0$ tali che

$0 \leq 2n^2 \leq cn^3$ per ogni $n \geq n_0$.

Per determinare c e n_0 scriviamo la disequazione $cn^3 - 2n^2 \geq 0$, ricavata dalla definizione di O applicata al nostro caso,

poichè $cn^3 - 2n^2 \geq 0 \Leftrightarrow$

$n^2 (cn - 2) \geq 0 \Leftrightarrow$

$n^2 \geq 0$ e $(cn - 2) \geq 0 \Leftrightarrow$

$cn \geq 2$

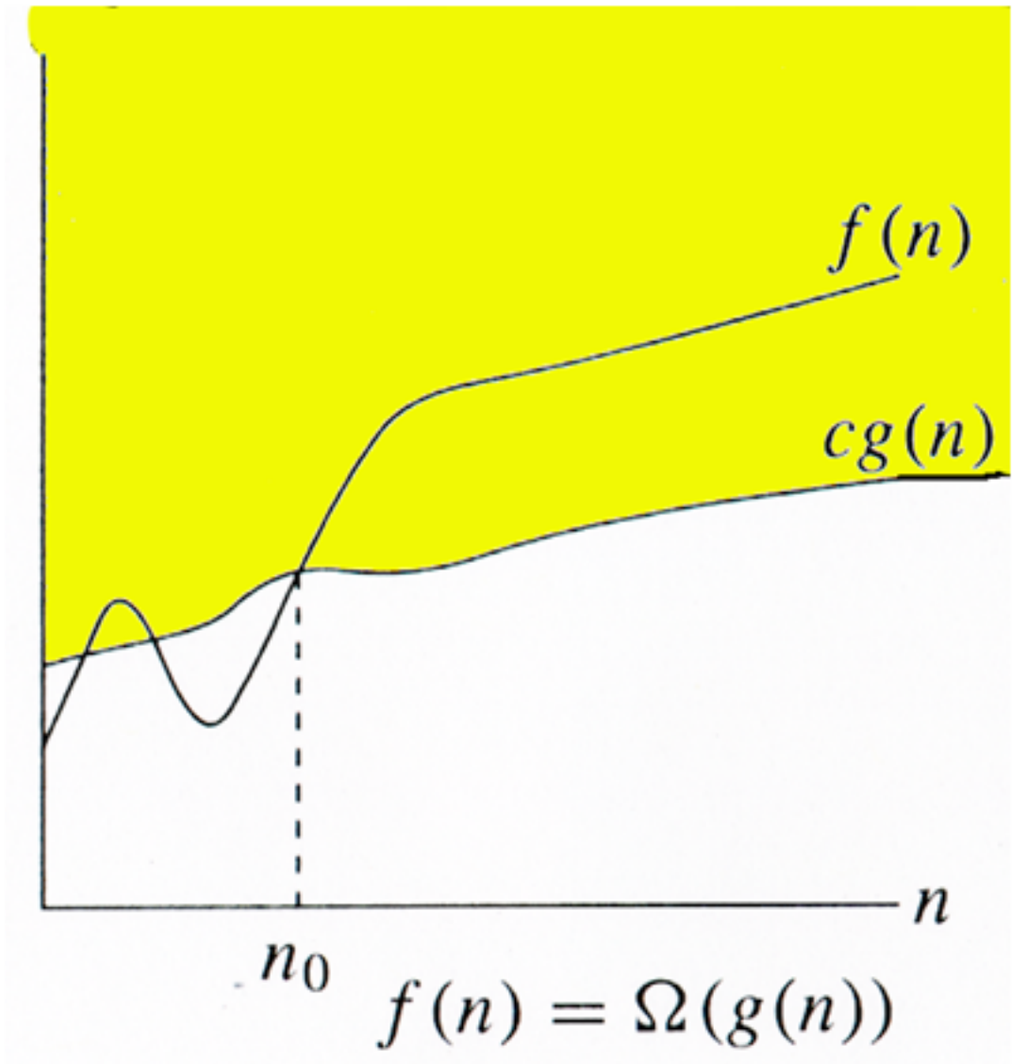
se prendiamo c = 2 sarà vero per ogni $n \geq 1$, per cui $n_0 = 1$

Ω grande

Analogamente a Θ e a O, Ω grande consente di eliminare le costanti, ma questa volta per esprimere un limite inferiore alla crescita della funzione

Ω(g(n)) = {f(n) | ∃ due costanti positive c e n_0 e f(n) ≥ c g(n) ≥ 0 per n ≥ n_0

Se f(n) = Ω(g(n)) allora g(n) è un limite inferiore asintotico per f(n)



Ω grande: esempi

$n^3 = \Omega(n^2)$
$n \neq \Omega(n^2)$

Funzioni in $\Omega(n)$:

n $n/1000$ $n^{1.999}$ n^2 n^2+n $1000n^2+50n$

Funzioni non in $\Omega(n)$:
ogni funzione costante
lg n

Ω grande: esempio di prova algebrica

Proviamo che $5n^3 = \Omega(n^2)$.

Per definizione di Ω devono esistere c e n_0 tali che $0 \leq c\, n^2 \leq 5n^3$ per ogni $n \geq n_0$.

Per determinare c e n_0

scriviamo la disequazione $5n^3 - cn^2 \geq 0$,
$n^2 (5n - c) \geq 0 \Leftrightarrow$

$n^2 \geq 0$ e $(5n - c) \geq 0$

se prendiamo c = 1 sarà vero per ogni n≥1, per cui $n_0 = 1$.

Esempio di uso di O e Ω grande

- **La ricerca sequenziale ha una complessità lineare, in Θ(n), nel caso peggiore, e costante, in Θ(1), nel caso migliore, la complessità di SeqS in tutti i casi è**

$$T_{SeqS}(n) = O(n) \text{ e } T_{SeqS}(n) = \Omega(1).$$

- **La ricerca binaria ha una complessità logaritmica, in Θ(lg n), nel caso peggiore e costante, in Θ(1), nel caso migliore, la complessità di BinS in tutti i casi è**

$$T_{BinS}(n) = O(\log n) \text{ e } T_{BinS}(n) = \Omega(1).$$

Θ grande, O grande e Ω grande

f(n) = O(g(n)) per dire che

g(n) è un limite superiore asintotico per f(n) o
che il tasso di crescita di g(n) è maggiore o uguale a quello di f(n)

f(n) = Ω(g(n)) per dire che

g(n) è un limite inferiore asintotico per f(n) o
che il tasso di crescita di g(n) è minore o uguale a quello di f(n)

f(n) = Θ(g(n)) per dire che

g(n) è un limite asintotico stretto per f(n)
o che il tasso di crescita di f(n) e g(n) è lo stesso

Notiamo infine che Ω è l'inverso di O infatti

g(n) = Ω(f(n)) sse f(n) = O(g(n)) e anche

g(n) = Θ(f(n)) sse g(n) = O(f(n)) e g(n) = Ω(f(n))

Complessità e Python

Per le operazioni su lista vedere:

http://interactivepython.org/runestone/static/pythonds/AlgorithmAnalysis/Lists.html

Uso dei limiti

Una funzione $g(n)$ è *asintoticamente nonnegativa* se $g(n) \geq 0$ per tutti gli $n \geq n_0$ per un certo $n_0 \in \mathbb{N}$

Se $f(n)$ e $g(n)$ sono funzioni *asintoticamente nonnegative*

allora possiamo dire che

se $\lim_{n\to\infty} f(n)/g(n) = < \infty$ (anche 0) allora $f(n) = O(g(n))$

se $\lim_{n\to\infty} f(n)/g(n) = c$ e $0 < c < \infty$ allora $f(n) = \Theta(g(n))$

se $\lim_{n\to\infty} f(n)/g(n) > 0$ (anche ∞) allora $f(n) = \Omega(g(n)))$

Qualche somma utile

- **Somma di potenze di interi successivi**

  $\sum_{i\in[0,n]} i^k = 1+2^k+3^k+\ldots+n^k = \Theta(n^{k+1})$

- **nel caso k=1 si ha la somma dei primi n numeri:**

  $\sum_{i\in[0,n]} i = 1+2+3+\ldots+n = n(n+1)/2 = \Theta(n^2)$

- **nel caso k = 2 si ha la somma dei primi quadrati:**

  $\sum_{i\in[0,n]} i^2 = 1+2^2+3^2+\ldots+n^2 = n(n+1)(2n+1)/6 = \Theta(n^3)$

Serie geometriche

Con $a \neq 1$ consideriamo $\sum_{i=0,\ldots,n} a^i = 1+a+a^2+\ldots+a^n$

se $a > 1$, $\sum_{i\in[0,n]} a^i = (a^{n+1} -1)/(a-1) = \Theta(a^n)$

nel caso a = 2 si ha somma delle prime potenze di 2:

$2^0 + 2^1 + 2^2 + \ldots + 2^n = 2^{n+1} - 1 = \Theta(2^n)$

se $0 < a < 1$, possiamo maggiorare la somma con la sua estensione all'infinito:

$\sum_{i\in[0,n]} a^i < \sum_{i\in[0,\infty)} a^i = 1/(1-a) = O(1)$.

nel caso a = 1/2 si ha somma:

$1 + 1/2 + 1/2^2 + \ldots + 1/2^n = 1/(1-1/2) = 2 = O(1)$

Altre somme utili

Infine consideriamo, con $a \neq 1$,

$a+2a^2+3a^3+\ldots+na^n = \sum_{i=1,\ldots,n} ia^i$

- se $a > 1$, $\sum_{i\in[0,n]} ia^i = a(1-(n+1)a^n+na^{n+1})/(1-a)^2 = O(na^n)$

nel caso $a = 2$ si ha somma:

$2 + 2*2^2 + 3*2^3 + \ldots + n*2^n = O(n2^n)$

- se $0 < a < 1$, anche la somma $\sum_{i\in[0,n]} ia^i$ può essere maggiorata con la sua estensione all'infinito:

$\sum_{i\in[0,n]} ia^i < \sum_{i\in[0,\infty]} ia^i = a/(1-a)^2 = O(1)$.

nel caso $a = 1/2$ si ha somma:

$1/2+ 2*(1/2)^2 +3*(1/2)^3 \ldots + n*(1/2)^n < 1/2/(1-1/2)^2 = 2 = O(1)$

Proprietà logaritmi e potenze

Definizione logaritmo: dati $b>0$, $b\neq1$ e $c>0$

$\log_b c$ = esponente da dare a b per avere c

quindi $\log_b b^n = n$ e $b = c^{\log_c b}$

- **Proprietà** dei **logaritmi (ben definiti)**
- $\log_b(xy) = \log_b x + \log_b y$

  $\log_b (x/y) = \log_b x - \log_b y$

  $\log_b (x^c) = c \log_b x$

  $\log_b c = \log_x c/\log_x b$

- **Proprietà** delle **potenze**

  $a^{(b+c)} = a^b a^c$

  $a^{bc} = (a^b)^c$

  $a^b/a^c = a^{(b-c)}$

  $b^c = a^{c*\log_a b}$

Esercizio 1 - notazione asintotica

Siano $g(n) = \lg n$ e $f(n) = \log_3 n^2$.
$f(n) = \Theta(\lg n)$?

Sì perchè $\log_3 n^2 = 2\log_3 n$ e sappiamo che $\log_3 n = \Theta(\lg n)$

Esercizio 2 - notazione asintotica

Siano $g(n) = \lg n$ e $f(n) = \log_3^2 n = (\log_3 n)^2$.
$f(n) = \Theta(\lg n)$?

No, perchè $(\log_3 n)^2 \neq O(\lg n)$.
Infatti $(\log_3 n)^2 = (\lg n/\lg 3)^2$ e dovremmo far vedere che esistono $c, n_0 \geq 0$ tali che $(\lg n/\lg 3)^2 \leq c \lg n$ per ogni $n \geq n_0$.
Ma $(\lg n/\lg 3)^2 \leq c \lg n \Leftrightarrow (\lg n)^2 \leq c(\lg 3)^2 \lg n \Leftrightarrow \lg n \leq c(\lg 3)^2$

Ma la funzione logaritmo non può essere limitata superiormente da una costante.
Infatti si può dimostrare che per ogni scelta di c si trova un valore di n che rende falsa la disuguaglianza.
Per esempio per ogni c, si può prendere $n = 2^{8c}$, allora $\lg n = 8c$ e $8c \not< c(\lg 3)^2$.

Esercizio 3 - notazione asintotica

Si determinino i termini dominanti nelle seguenti espressioni, che possiamo supporre derivanti dall'analisi di algoritmi per problemi su n elementi, e li si utilizzino per esprimere un limite superiore alla crescita asintotica dell'espressione stessa:

espressione	termine dominante	O(…)
$500n + 100n^{1,5} + 50n\log_{10}n$	$100n^{1,5}$	$O(n^{1,5})$
$3n^2\lg n + 4n(\lg n)^2$	$3n^2\lg n$	$O(n^2\lg n)$
$n\log_3 n + n\lg n$	$n\lg n$	$O(n \lg n)$
$100n\log_3 n + 5n^3 + 100n$	$5n^3$	$O(n^3)$

Nota che $n^{1,5} = n^{3/2} = (\sqrt{n})^3$

Esercizi

Es. Notazione asintotica:

1. Si dimostri che $f(n)+g(n) = \Theta(\max\{f(n),g(n)\})$ sotto l'ip. $f(n),g(n) > 0$, a partire da un certo n_0.

2. Si dimostri che se $f(n) = \Theta(n^k)$, per una costante k, cioè f(n) è polinomiale di grado k, allora $\lg(f(n)) = \Theta(\lg n)$, dove $\lg(x) = \log_2(x)$.

3. Si dimostri, usando i limiti, che $p(n) = a_d n^d + a_{d-1} n^{d-1} + \ldots + a_1 n + d = \Theta(n^d)$, se $a_d > 0$

Es. Analisi

1. Dato un array X di n elementi, un algoritmo D, che riceve in input X, esegue un calcolo di complessità $\Theta(n)$ per ogni numero pari in X, e uno di complessità $\Theta(\log n)$ per ogni numero dispari. Qual'è la complessità di tempo di D nei casi migliore e peggiore?