Notazione asintotica
II parte

Notazione asintotica
limite inferiore e superiore

- Date $f : n \in N \to f(n) \in R^+$, $g : n \in N \to g(n) \in R^+$,
  $f(n) = O(g(n)) \Leftrightarrow g(n)=\Omega(f(n))$

Infatti, applicando la definizione di O

$f(n) = O(g(n)) \Leftrightarrow \exists\, c > 0$ ed $\exists\, n_0$ tali che $f(n) \leq cg(n)$, $\forall\, n \geq n_0$

cio` equivale a scrivere

$\exists\, c > 0$, $\exists\, n_0$ tali che $g(n) \geq (1/c)f(n)$, $\forall\, n \geq n_0$

e cio` equivale a scrivere $g(n)=\Omega(f(n))$

Notazione asintotica
limite inferiore e superiore

- Esempio
- $\log^a n^b = O(n^k)$ equivale a scrivere $n^k = \Omega(\log^a n^b)$

Notazione asintotica Ω

- Se dimostriamo che un certo algoritmo ha tempo di esecuzione $\Omega(n)$ nel caso ottimo allora possiamo dire che il tempo di esecuzione dell'algoritmo è $\Omega(n)$ per ogni input

Notazione asintotica Ω

InsertionSort(a):	Costo	Num. Volte
FOR(i=1;i<n;i=i+1){	c_1	n
elemDaIns=a[i];	c_2	n-1
j=i;	c_3	n-1
while((j>0)&& a[j-1]>elemDaIns){	c_4	$\sum_{i=1}^{n-1} t_i$
a[j]=a[j-1];	c_5	$\sum_{i=1}^{n-1} (t_i - 1)$
j=j-1;	c_6	$\sum_{i=1}^{n-1} (t_i - 1)$
}		
a[j]=elemDaIns;	c_7	n-1
}		

t_i è il numero di iterazioni del ciclo di while all'i-esima iterazione del for

Notazione asintotica Ω

- Tempo di esecuzione di InsertionSort:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1} t_i + c_5 \sum_{i=1}^{n-1} (t_i - 1) + c_6 \sum_{i=1}^{n-1} (t_i - 1) + c_7(n-1)$$

con $c_i \geq 1$ per ogni i

- il caso ottimo si verifica quando l'array input è già ordinato. In questo caso $t_i = 1$

- Tempo di esecuzione nel caso ottimo:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$
$$= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) = \Omega(n)$$

Notazione asintotica Ω

- InsertionSort nel caso ottimo ha tempo di esecuzione $\Omega(n)$
- Ne consegue che InsertionSort ha tempo di esecuzione $\Omega(n)$ per tutti gli input
- Abbiamo dimostrato che InsertionSort nel caso pessimo ha tempo di esecuzione $O(n^2)$
- In generale, possiamo dire che il tempo di esecuzione di InsertionSort è compreso tra $\Omega(n)$ e $O(n^2)$

Notazione asintotica Ω

- La notazione Ω può essere usata per limitare inferiormente la complessità di un problema computazionale (da non confondere con quella dell'algoritmo!)
- Per dire che un problema ha complessità $\Omega(f(n))$ occorre dimostrare che qualsiasi algoritmo richiede tempo $\Omega(f(n))$ per risolvere una generica istanza di dimensione n del problema

Notazione asintotica

- Sia P un problema computazionale che ha complessità $\Omega(f(n))$

  → Un algoritmo A che risolve P in tempo $O(f(n))$ è un algoritmo **ottimo**

Notazione asintotica Ω

- Esempio dell'uso di Ω nella limitazione inferiore della complessità di un problema:
- Il problema del massimo:
  - Dato un array di n numeri vogliamo trovare il numero massimo contenuto nell'array
  - Occorrono almeno n-1 confronti
  - Intuizione: ogni numero diverso dal massimo deve "perdere" in almeno un confronto

  → il problema del massimo ha complessità Ω(n)

Esempio di algoritmo ottimo

L'algoritmo Max è ottimo perché ha tempo di esecuzione O(n) (effettua esattamente n-1 confronti)

Tempo di esecuzione di Max(a):

$T(n)=c_1+c_2n+(c_3+c_4)(n-1)+c_5=(c_2+c_3+c_4)n+c_1-c_3-c_4+c_5=an+b=O(n)$

- n è la lunghezza dell'array

Max(a):	Costo	num. volte
max=a[0];	c_1	1
FOR(i=1;i<n;i++)	c_2	n
IF(a[i]>max)	c_3	n-1
max=a[i];	c_4	n-1 (nel caso pessimo)
RETURN max;	c_5	1

Notazione asintotica Ω

- Siamo interessati ora a capire quanto e` buona la nostra stima del tempo di esecuzione di InsertionSort nel caso pessimo

- Abbiamo dimostrato che il tempo di esecuzione di InsertionSort nel caso pessimo è $T(n)=an^2+bn+c$, con $a>0$

- Abbiamo dimostrato che $T(n)=an^2+bn+c$ e` sia **$O(n^2)$** che **$\Omega(n^2)$**

- **Quindi il tempo di esecuzione nel caso pessimo T(n) di InsertionSort e` sia $O(n^2)$ che $\Omega(n^2)$.**

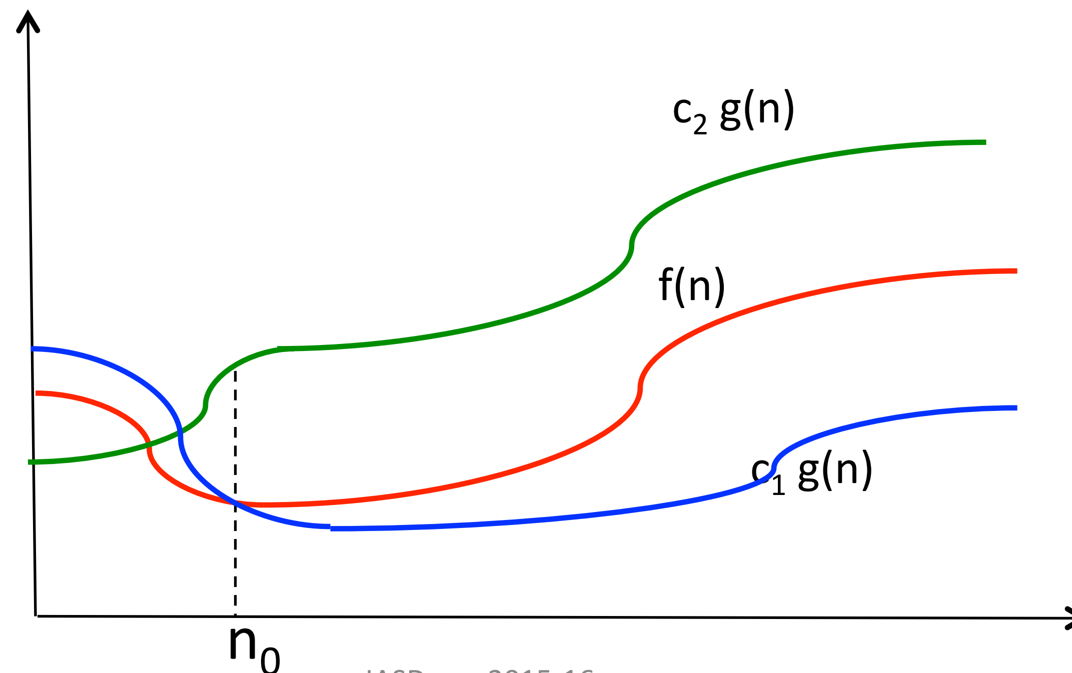
- **Diremo che il tempo di esecuzione di InsertionSort nel caso pessimo e` $\Theta(n^2)$**

Notazione asintotica
limite stretto

- Date $f : n \in N \rightarrow f(n) \in R^+$, $g : n \in N \rightarrow g(n) \in R^+$, scriveremo $f(n) = \Theta(g(n))$

$\Leftrightarrow \exists\, c_1, c_2 > 0, \exists\, n_0$ tale che $c_1 g(n) \leq f(n) \leq c_2 g(n), \;\forall\, n \geq n_0$

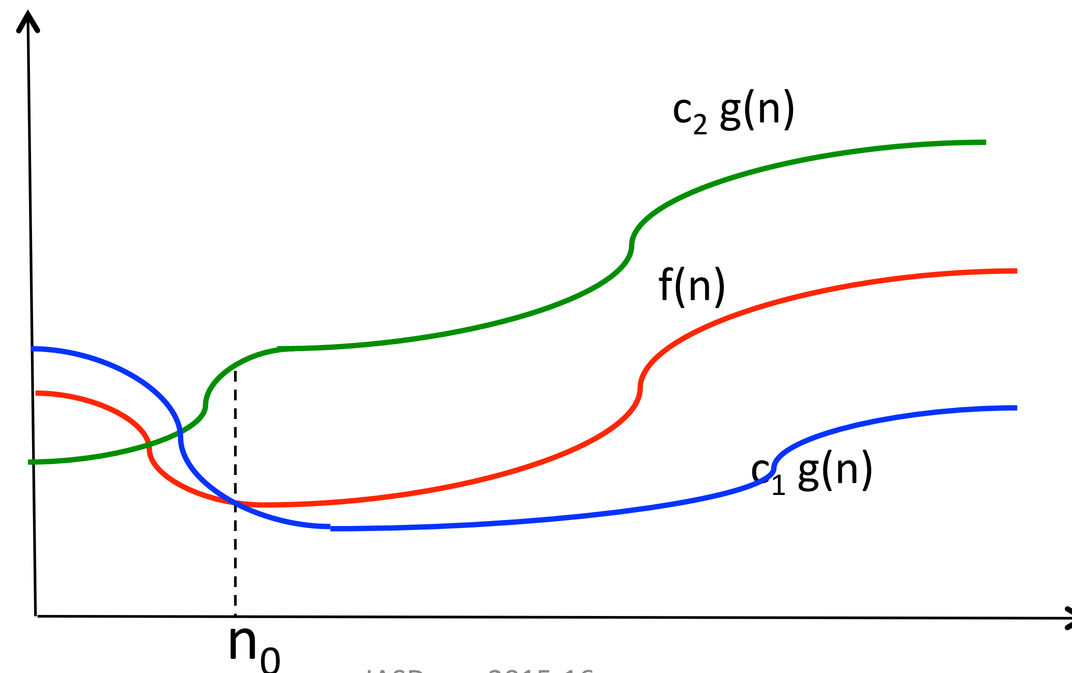
Informalmente, $f(n) = \Theta(g(n))$ se $f(n)$ cresce come $g(n)$.



Notazione asintotica
limite stretto

- Teorema: Date due funzioni f(n) e g(n),

$$f(n) = \Theta(g(n)) \Leftrightarrow f(n)=\Omega(g(n)) \text{ e } f(n)= O(g(n))$$



Ordine di grandezza dei polinomi

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 = \Theta(n^k), \quad a_k > 0$$

Infatti abbiamo dimostrato che

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 = O(n^k)$$

e

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 = \Omega(n^k), \quad a_k > 0$$

Notazione asintotica
limite stretto

- Possiamo dire che $\log n = \Theta(n)$? Ovviamente **NO**!
- E` evidente che $\log n$ cresce piu` lentamente di cn per ogni costante $c>0$ e quindi $\log n$ **non** e` $\Omega(n)$
- In modo un po` piu` formale…
- Vediamo cosa succede quando proviamo a determinare due costanti $c>0$ ed n_0 tali che $\log n \geq cn$, $\forall n \geq n_0$
- $\log n \geq cn$ se e solo se $c \leq (\log n)/n$
- La volta scorsa abbiamo dimostrato $\log n \leq (\sqrt{n})/2$ per ogni $n \geq 1$ (caso $b=1$ e $k=1/2$) e quindi $(\log n)/n \leq (\sqrt{n})/(2n) = 1/(2\sqrt{n})$ per ogni $n \geq 1$
- $1/(2\sqrt{n})$ diventa piu` piccolo man mano che cresce n per cui qualsiasi sia la costante c che scegliamo esiste un valore di n abbastanza grande per cui $c > 1/(2\sqrt{n})$ (basta prendere $n > 1/(2c^2)$)

Complessità di un problema computazionale

- Se un problema ha complessità $\Omega(f(n))$ ed esiste un algoritmo A che lo risolve e che ha tempo di esecuzione $O(f(n))$ allora il problema ha complessità $\Theta(f(n))$

Utili regole per la notazione asintotica

- Moltiplicazione per una costante
  - $g(n)= O(f(n)) \rightarrow a \cdot g(n)=O(f(n))$, per ogni constante a
  - Esempio:
    - $\log n=O(n) \rightarrow 5\log n=O(n)$
- Somma di funzioni
  - $g(n)=O(f(n))$ e $q(n)=O(h(n)) \rightarrow g(n)+q(n)=O(f(n)+h(n))=O(\max\{f(n),h(n)\})$
  - Esempio:
    - $n+\log(n)=O(n)$ e $4n^2=O(n^2)$

    $\rightarrow 4n^2+n+\log n=O(n^2+n)=O(n^2)$

Utili regole per la notazione asintotica

- Moltiplicazione di funzioni
  - $g(n)=O(f(n))$ e $q(n)=O(h(n))$ ➔ $g(n)\cdot q(n)=O(f(n)\cdot h(n))$
  - Esempio:
    - $n+\log(n)=O(n)$ e $4n^2=O(n^2)$ ➔ $4n^2(n+\log n)=O(n^2 \cdot n)=O(n^3)$
- Transitività
  - $g(n)= O(f(n))$ e $f(n)=O(h(n))$ ➔ $g(n)=O(h(n))$
  - Esempio:
  - $\log(n)=O(n^{1/3})$ e $n^{1/3}=O(n^{1/2})$ ➔ $\log(n)=O(n^{1/2})$

Dimostrazione transitivita`

$g(n)=O(f(n))$ e $f(n)=O(h(n)) \rightarrow g(n)=O(h(n))$

Dimostrazione:

$g(n)=O(f(n)) \leftrightarrow$ esistono $c_1>0$ ed n_1 t.c. $g(n)\leq c_1 f(n)$ per ogni $n\geq n_1$

$f(n)=O(h(n)) \leftrightarrow$ esistono $c_2>0$ ed n_2 t.c. $f(n)\leq c_2 h(n)$ per ogni $n\geq n_2$

$\rightarrow g(n) \leq c_1 f(n) \leq c_1 c_2 h(n)$ per ogni $n\geq \max\{n_1,n_2\}$

Tempo esponenziale $O(c^n)$

- Esempio: Sudoku
- Input:
  - Tabella 9 x 9 contenente numeri 1,2,…,9
  - Divisa in 9 sottotabelle di dimensione 3 x3
  - Alcune celle contengono numeri mentre altre sono vuote
- Output:
  - Celle vuote riempite in modo che
    - ciascuna riga contiene una permutazione di 1,2,...,9
    - ciascuna colonna contiene una permutazione di 1,2,...,9
    - ciascuna sottotabella contiene una permutazione di 1,2,...,9

Sudoku: backtrack

- Un algoritmo di backtrack per il gioco del Sudoku consiste nel riempire le caselle vuote in tutti i modi possibili (facendo scelte ammissibili rispetto alle celle già riempite) fino a che non si arriva ad una soluzione.
- L'algoritmo fa "backtrack" quando si accorge che le scelte effettuate fino a quel momento non possono portare ad una soluzione.
  - L'algoritmo annulla l'ultima scelta fatta: torna nell'ultima cella in cui ha inserito un numero e sceglie per quella cella un altro valore ammissibile. Se per quella cella sono stati già esplorati tutti i valori possibili, l'algoritmo torna alla cella precedente.

Sudoku: backtrack

- Se ci sono k ($k \leq 9^2$) celle vuote, nel caso pessimo vengono esaminate 9^k scelte
- In generale, per una tabella n x n si eplorano fino a $n^{n^2} = 2^{\log(n)n^2}$ scelte

- Esiste un algoritmo polinomiale per il sudoku?
  – Risposta: Ad oggi non è noto

Problemi trattabili

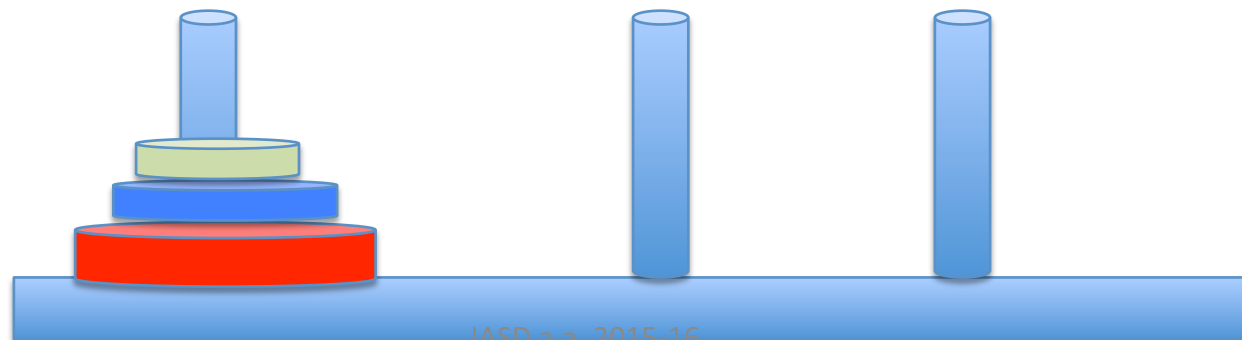
- Problemi per cui esiste un algoritmo polinomiale che li risolve

- Sudoku è trattabile?
  - Risposta: Non si sa

Problemi intrattabili

Esempio: Le torri di Hanoi

- 3 pioli
- n = 64 dischi sul primo (vuoti gli altri due)
- disco più grande non può stare su più piccolo
- ogni mossa sposta un disco
- Obiettivo: spostarli tutti dal primo all'ultimo piolo



Algoritmo ricorsivo per il problema delle Torri di Hanoi

```
TorriHanoi(n , primo, secondo, terzo):
 IF(n=1){
   primo → terzo;
} ELSE{
  TorriHanoi(n-1,primo, terzo,secondo);
   primo→terzo;
   TorriHanoi(n-1,secondo, primo,terzo);
 }
```

Quante mosse fa l'algoritmo TorriHanoi?

Numero di mosse = 2^n-1

Dimostrazione per induzione su n

- Caso base n = 1:
  - numero mosse: $2^1-1 = 1$
- Passo induttivo: supponiamo che per n-1 si facciano 2^{n-1} -1 mosse
  - numero mosse per n: $(2^{n-1}-1)+1+(2^{n-1}-1) = 2^n-1$

n=64: $2^{64}-1$ = 18 446 744 073 709 551 615

Se 1 mossa/sec → circa 585 miliardi di anni!

Tempo algoritmo TorriHanoi

- Esempio: se assumiamo che una mossa richieda 1 sec
  - 5 dischi in 31 sec
  - 10 dischi in 17 min
  - 45 dischi in 1115689 a

Abbiamo visto che 1 mossa/sec → $t=2^n-1$ secondi per n dischi

Se invece di fare una mossa al secondo, si fanno b mosse al secondo allora nel tempo t riesco a risolvere il gioco per $n+m$ dischi, dove $t=(2^{n+m}-1)/b$

→ $n+m = \log_2(tb+1) \sim \log_2 t + \log_2 b \sim n + \log_2 b$

→Aumentare il numero di operazioni svolte in un secondo di un fattore **moltiplicativo** b migliora **solo** di un fattore **additivo** $\log_2 b$ il numero di dischi

Esempio: 1 mossa/sec → 64 dischi in $2^{64}-1$ secondi

2^{30} mosse/sec → 94 dischi in $2^{64}-1$ secondi

Problemi intrattabili

- Il problema delle Torri di Hanoi è intrattabile?
  - Risposta: sì
  - E` stato dimostrato che non è possibile usare meno di 2^n-1 mosse (non può esistere un algoritmo che usi un numero inferiore di mosse)

Tempo polinomiale

Torri di Hanoi generalizzate con k > 3 pioli

- pioli numerati da 0 a k-1
- ipotesi semplificativa: n è multiplo di k-2

```
TorriHanoiGen(n,k)
 FOR(i=1;i<=k-2;i=i+1)
   TorriHanoi((n/(k-2)), 0,k-1,i);
 FOR(i=k-2;i>0;i=i-1)
    TorriHanoi((n/(k-2)), i,0,k-1);
```

Torri di Hanoi generalizzate

- Il codice richiede 2 x (k-2) x ($2^{n/(k-2)}$-1) mosse
- Fissando k =$\lceil n/\log n \rceil + 2$

$$2(k-2)(2^{n/(k-2)}-1) < 2(n/\log n+1)(2^{n/(n/\log n)}-1) =$$

$$2(n/\log n+1)(2^{\log n}-1) = 2(n/\log n+2)(n-1) = O(n^2)$$

Dove reperire gli argomenti introduttivi

http://wps.pearsoned.it/wps/media/objects/14141/14480998/calcolabilita_ecomplessita_.pdf