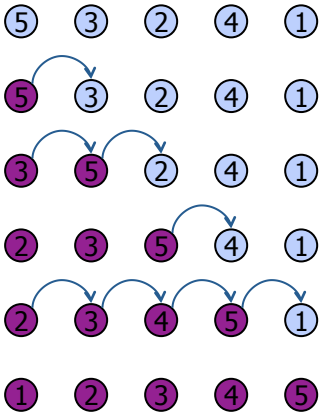
Notazione asintotica

Analisi degli algoritmi

- Esempio:

```
InsertionSort(a):       //n e` la lunghezza di a
 FOR(i=1;i<n;i=i+1){
  elemDaIns=a[i];
  j=i;
  while((j>0)&& a[j-1]>elemDaIns){ //cerca il posto per a[i]
       a[j]=a[j-1];     //shifto a destra gli elementi piu` grandi
       j=j-1;
}
 a[j]=elemDaIns;
}
```

Insertion-sort



Analisi di InsertionSort

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

Analisi di InsertionSort

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

- Nel caso pessimo t_i = i +1 (elementi in ordine decrescente ➔tutti gli elementi che precedono a[i] sono più grandi di a[i])

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

Analisi di InsertionSort caso pessimo

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

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

$$= (c_1 + c_2 + c_3 + c_7)n - c_2 - c_3 - c_7 + (c_4 + c_5 + c_6)\left(\frac{n(n-1)}{2}\right) + c_4(n-1)$$

$$= (c_4 + c_5 + c_6)\frac{n^2}{2} + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right)n - c_2 - c_3 - c_4 - c_7$$

$$= an^2 + bn + c$$

Ordine di grandezza

- Nell'analizzare la complessità di InsertionSort abbiamo operato delle astrazioni
  - Abbiamo ignorato il valore esatto prima delle costanti c_i e poi delle costanti a, b e c.

Ordine di grandezza

- Possiamo aumentare il livello di astrazione considerando solo l'ordine di grandezza
  - Consideriamo solo il temine "dominante"
    - Per InsertionSort: an^2
    - Giustificazione: più **grande** è n, minore è il contributo dato dagli altri termini alla stima della complessità
  - Ignoriamo del tutto le costanti
    - Diremo che InsertionSort ha complessità $\Theta(n^2)$
    - Giustificazione: più grande è n, minore è il contributo dato dalle costanti alla stima della complessità

Efficienza asintotica degli algoritmi

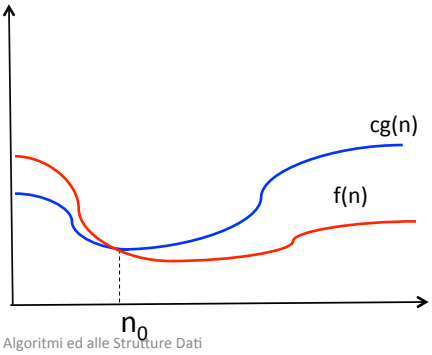
- Per input piccoli può non essere corretto considerare solo l'ordine di grandezza ma per input "abbastanza" grandi è corretto farlo

Notazione asintotica

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

$\Leftrightarrow \exists\, c > 0, \exists\, n_0$ tale che $f(n) \leq cg(n), \forall\, n \geq n_0$

Informalmente, $f(n) = O(g(n))$ se $f(n)$ non cresce più velocemente di $g(n)$.



Notazione asintotica

- **Esempio**

- $10n^3 + 2n^2 + 7 = O(n^3)$
- Dimostriamolo:
  - Occorre provare che

    $\exists\, c, n_0 : 10n^3 + 2n^2 + 7 \le cn^3, \ \forall\, n \ge n_0$
  - Si ha: $10n^3 + 2n^2 + 7 \le 10n^3 + 2n^3 + 7$

    $\le 10n^3 + 2n^3 + n^3 = 13n^3, \ \forall\, n \ge 2.$
  - Quindi la diseguaglianza è soddisfatta per c = 13 e $n_0 = 2$.

Ordine di grandezza dei polinomi

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

Infatti

$$\begin{aligned} a_k n^k + a_{k-1} n^{k-1} \quad + \quad & \cdots + a_1 n + a_0 \\ \le \ & |a_k| n^k + |a_{k-1}| n^{k-1} + \cdots + |a_1| n + |a_0| \\ \le \ & |a_k| n^k + |a_{k-1}| n^k + \cdots + |a_1| n^k + |a_0| n^k \\ = \ & (|a_k| + |a_{k-1}| + \cdots + |a_1| + |a_0|) n^k \\ = \ & c n^k \end{aligned}$$

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

Esempi

$n^3 + 100n + 200 = O(n^3)$

$20n^3 + n^5 + 100n = O(n^5)$

$10n^2 + n^{5/2} + 7n = O(n^{5/2})$

$10n+3n^7+5n^6+9n^3+34n^2+22n^5+n^{8/3}+4n^{7/2}+23n^{11/2} = O(n^7)$

Tempo lineare O(n)

Esempio: L'algoritmo Max cerca il massimo nell'rray a

- 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

Tempo lineare O(n)

Esempio: L'algoritmo Merge(a,b,c) prende in input due array a e b di numeri ordinati e un array c vuoto e inserisce in c gli elementi di a e b in modo che c risulti ordinato.

```
Merge(a,b,c): //la lunghezza di a  è m e quella di b è q
 i=0;    //indice di a
 j=0;   //indice di b
 k=0; //indice di c
WHILE(i<m && j<q){              //al massimo m+q=n iterazioni
  IF(a[i]<b[j]) { c[k]=a[i]; i=i+1;}  //perche` ad ogni iterazione
  ELSE    { c[k]=b[j]; j=j+1;}       //incremento i o j
  k=k+1;
}
WHILE(i<m) { c[k]=a[i]; i=i+1;k=k+1;}
WHILE(j<q) { c[k]=b[j]; j=j+1;k=k+1;}
}
```

Tempo di esecuzione T(n)=O(n)

Tempo quadratico $O(n^2)$

- Esempio: L'algoritmo Intersect inserisce nell'array c gli elementi che compaiono sia in a che in b. Si assuma che gli elementi in ciascuno degli array a e b siano distinti e che entrambi gli array a e b abbiano dimensione uguale ad n.

```
Intersect(a,b,c):
 k=0;
 FOR(i=0;i<n;i++){              //  eseguito n+1 volte
    FOR(j=0;j<n;j++){           //  eseguito (n+1)^2 volte
      IF(a[i]=b[j])             //    eseguito n^2 volte
         {c[k]=a[i];k=k+1;}  //eseguito n volte (nel caso pessimo)
```

Tempo di esecuzione $T(n)=O(n^2)$

Tempo logaritmico O(log n)

- Esempio: Algoritmo di ricerca binaria: cerca un numero x in un array a ordinato

```
RicercaBinaria(a,x)
primo=0;
ultimo=n-1;
WHILE(primo<ultimo)   //al massimo ⌈log(n)⌉+1 volte
  centro=(primo+ultimo)/2; //il corpo al massimo ⌈log(n)⌉ volte
  IF( x< centro)  ultimo=centro-1;
    ELSE IF( x> centro)  primo=centro+1;
  ELSE RETURN centro;
}
```

Tempo di esecuzione T(n)=O(log n)

Ordine di grandezza del logaritmo

- Proviamo che

$$\log_2 n = O(n).$$

Occorre provare che $\exists c, n_0 \ : \ \log_2 n \leq cn \quad \forall n \geq n_0$

Per induzione su n: Per $n = 1$ abbiamo $\log_2 1 = 0 \leq 1$.

In generale, per $n \geq 1$

$$\begin{aligned} \log_2(n+1) &\leq \log_2(n+n) = \log_2(2n) \\ &= \log_2 2 + \log_2 n = 1 + \log n \\ &\leq 1 + n \text{ (per ipotesi induttiva)} \end{aligned}$$

Abbiamo quindi provato che

$$\log n \leq n \quad \forall n \geq 1 \Longrightarrow \boxed{\log n = O(n)}$$

- E se il logaritmo non è in base 2?

Non cambia niente. Sappiamo infatti che

$$\log_a n = (\log_a 2)(\log_2 n),$$

per $a \neq 1 \neq b$, inoltre abbiamo già provato che $\log_2 n \leq n$, pertanto

$$\log_a n = (\log_a 2)(\log_2 n) \leq (\log_a 2)n \;\Rightarrow\; \log_a n = O(n)$$

E se invece di $\log n$ abbiamo $\log n^a,\ a > 1$? Non cambia niente. Sappiamo che $\log n^a = a \log n$ e $\log n \leq cn$, per opportuna costante $c > 0$, quindi

$$\log n^a = a \log n \leq acn \quad \text{ovvero} \quad \log n^a = O(n)$$

Logaritmo al confronto con radice

- Si ha

$$\boxed{\forall \text{ costanti } a > 0, b > 0, k > 0 \text{ vale } \log^a n^b = O(n^k)} \quad (2)$$

Ad esempio, $\log^5 n^6 = O(\sqrt[3]{n})$

Basta infatti porre $a = 5, b = 6, k = 1/3$ nella (2)

Altro esempio: $\log^2 n^{10} = O(\sqrt[6]{n})$

Basta porre $a = 2, b = 10, k = 1/6$ nella (2)

Occorre provare che per ogni $a, b, k > 0$

$$\exists c, n_0 \ : \ (\log n^b)^a \leq cn^k, \ \forall n \geq n_0 \qquad (3)$$

Proviamolo innanzitutto nel caso $a = 1$. Ricordiamo che $\log x^y = y \log x$, e che $\log x \leq dx$. Pertanto

$$(\log n^b) = (b \log n) = (b\frac{1}{k} \cdot k \log n) = (b\frac{1}{k} \cdot \log n^k) \leq b\frac{1}{k} d n^k,$$

provando la (3) per $a = 1$, (ma $\forall k$), con $c = (bd)/k$. Nel caso generale, usando la (3) con parametri $a = 1, b$ arbitrario, e k/a avremo $\log n^b \leq cn^{k/a}$

- $\log^a n^b = (\log n^b)^a \leq (cn^{k/a})^a = c^a n^k$
- c^a è la costante

Espressione O	nome
$O(1)$	costante
$O(\log \log n)$	$\log \log$
$O(\log n)$	logaritmico
$O(\sqrt[c]{n}), \ c > 1$	sublineare
$O(n)$	lineare
$O(n \log n)$	$n \log n$
$O(n^2)$	quadratico
$O(n^3)$	cubico
$O(n^k) \ (k \geq 1)$	polinomiale
$O(a^n) \ (a > 1)$	esponenziale

Utili regole per la notazione asintotica

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

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 $n^{1/3}=O(n^{1/2})$ ➔$\log(n)=O(n^{1/2})$

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 backtrack?
  - Risposta: Ad oggi non è noto

Problemi trattabili

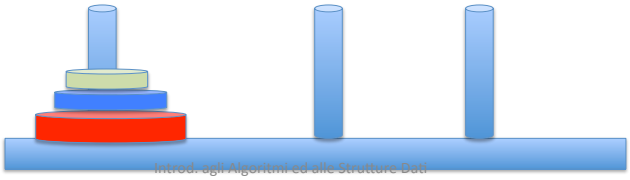
- Problemi per cui esiste un algoritmo polinomiale che li risolve

- 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

Se invece di fare una mossa al secondo, si fanno b mosse al secondo 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

10^9 mosse/sec → 93 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
- Al più n^2 mosse, fissando k = $\lfloor n/\log n \rfloor$ e n≥5
- n = 64: al più 64^2 = 4096 mosse

```
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);
```


Tempo polinomiale n^2

- Esempio: Se assumiamo di fare 1 mossa/secondo
- 5 dischi in 25 sec
- 10 dischi in 100 sec
- 45 dischi in 34 min
- Se invece di fare una mossa al secondo, si fanno b mosse al secondo nel tempo t riesco a risolvere il gioco per n+m dischi, dove $t=(n+m)^2/b$
  → $n+m = (tb)^{1/2} = \sqrt{t}\sqrt{b}=n\sqrt{b}$
- Aumentare di un fattore **moltiplicativo** b (ossia b operazioni/sec) aumenta il numero di dischi di un fattore **moltiplicativo** $\sqrt{b}$

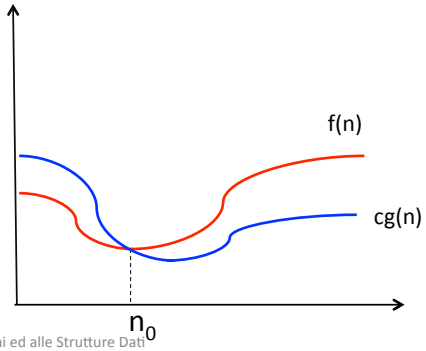
Esempio: 1 mossa/sec → 64 dischi in 64^2 secondi
10^9 mosse/sec → 2023857 dischi in 64^2 secondi

Notazione asintotica
limite inferiore

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

$\Leftrightarrow \exists\, c > 0, \exists\, n_0$ tale che $f(n) \geq cg(n), \forall\, n \geq n_0$

Informalmente, $f(n) = \Omega(g(n))$ se $f(n)$ non cresce più lentamente di $g(n)$.



Notazione asintotica
Limite inferiore

- **Esempio**

- $10n^3 - 2n^2 + 7 = \Omega(n^3)$
- Dimostriamolo:
  - Occorre provare che

    $\exists\, c, n_0 : 10n^3 - 2n^2 + 7 \geq cn^3, \forall\, n \geq n_0$
  - Diviamo tutto per n^3: $10 - 2/n + 7/n^3 \geq c$
  - $10 - 2/n + 7/n^3 > 10 - 2 = 8, \forall\, n \geq 1$.
  - Quindi la diseguaglianza è soddisfatta per $c \leq 8$ e $n_0 = 1$.

Notazione asintotica Ω

- Se dimostriamo che un certo algoritmo ha tempo di esecuzione Ω(n) nel caso ottimo allora possiamo dire che il tempo di esecuzione dell'algoritmo è Ω(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:

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

Per dimostrare che è Ω(n) basta prendere $c < c_1+c_2+c_3+c_4+c_7$ e $n_0 = c_2+c_3+c_4+c_7$

Notazione asintotica Ω

- InsertionSort nel caso ottimo ha tempo di esecuzione Ω(n)
- Ne consegue che InsertionSort ha tempo di esecuzione Ω(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 Ω(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 Ω (f(n)) occorre dimostrare che qualsiasi algoritmo richiede tempo Ω (f(n)) per risolvere una generica istanza di dimensione n del problema

Notazione asintotica

- Sia P è un problema computazionale che ha complessità Ω (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

Ordine di grandezza dei polinomi

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

Dim. Vogliamo trovare c ed n_0 tali che

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \geq c n^k, \ \forall n \geq n_0$$

Diviamo tutto per n^k

$$a_k + \frac{a_{k-1}}{n} + \cdots + \frac{a_i}{n^{k-i}} + \cdots + \frac{a_1}{n^{k-1}} + \frac{a_0}{n^k} \geq c$$

Consideriamo la parte sinistra. Si ha

$$\begin{aligned} & a_k + \frac{a_{k-1}}{n} + \cdots + \frac{a_i}{n^{k-i}} + \cdots + \frac{a_1}{n^{k-1}} + \frac{a_0}{n^k} \\ \geq \quad & a_k - \frac{|a_{k-1}|}{n} - \cdots - \frac{|a_i|}{n^{k-i}} + \cdots - \frac{|a_1|}{n^{k-1}} - \frac{|a_0|}{n^k} \end{aligned}$$

Ordine di grandezza dei polinomi

Si ha che $\frac{|a_i|}{n^{k-i}} \leq \frac{a_k}{k+1}$ per $n \geq (\frac{|a_i|(k+1)}{a_k})^{\frac{1}{k-i}}$.

Quindi per $n \geq \max_{0 \leq i \leq k-1} \left\{ \left(\frac{|a_i|(k+1)}{a_k} \right)^{\frac{1}{k-i}} \right\}$ si ha

$$a_k - \frac{|a_{k-1}|}{n} - \cdots - \frac{|a_i|}{n^{k-i}} + \cdots - \frac{|a_1|}{n^{k-1}} - \frac{|a_0|}{n^k} \geq a_k - k\frac{a_k}{k+1} > 0.$$

Ne consegue che possiamo scegliere

$$0 < c \leq a_k - \frac{k}{k+1} a_k, \quad n_0 = \max_{0 \leq i \leq k-1} \left\{ \left(\frac{|a_i|(k+1)}{a_k} \right)^{\frac{1}{k-i}} \right\}$$

Ordine di grandezza dei polinomi

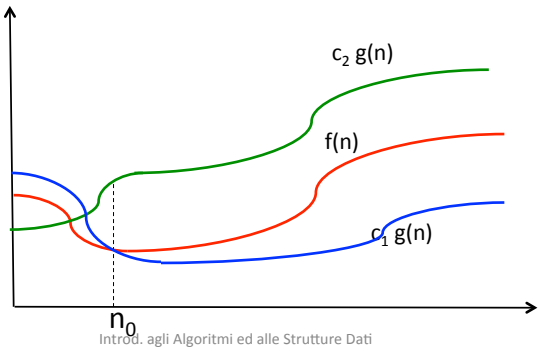
- Esempio: InsertionSort nel caso pessimo
- In una delle prime lezioni abbiamo dimostrato che il tempo di esecuzione di InsertionSort nel caso pessimo è $T(n)=an^2+bn+c$, con $a>0$
- Ne consegue che **$T(n)=\Omega(n^2)$** nel **caso pessimo**

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 $c_1 g(n) \leq f(n) \leq c_2 g(n), \ \forall\, n \geq n_0$

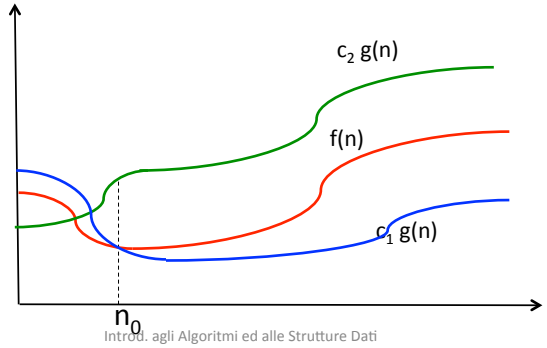
Informalmente, $f(n) = \Theta(g(n))$ se $f(n)$ non cresce come $g(n)$.



Notazione asintotica
limite stretto

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

  $f(n) = \Theta(g(n)) \Leftrightarrow f(n)=\Omega(g(n))$ 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

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

Complessità di un problema computazionale

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

Dove reperire gli argomenti introduttivi

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