Fondamenti

Problemi

problèma s. m. [dal lat. *problema -ătis* «questione proposta», gr. πρόβλημα -ατος, der. di προβάλλω «mettere avanti, proporre»] (pl. -*i*).

– 1. Ogni quesito di cui si richieda ad altri o a sé stessi la soluzione, partendo di solito da elementi noti.

In partic.: a. In matematica, quesito che richiede la determinazione o la costruzione di uno o più enti (numeri, funzioni, figure geometriche, insiemi, ecc.) che soddisfino alle condizioni specificate nell'enunciato del problema

http://www.treccani.it/vocabolario/problema/

Problemi

Un po' più nello specifico:

- Ci vengono forniti dei dati in input, sappiamo come sono strutturati (es. "una lista di interi", "un grafo pesato")
- Ci vengono richiesti dei dati in uscita, non noti ma che *devono soddisfare delle condizioni note* (es. "il numero più grande", "una componente connessa").

Dati in input ⇨

⇨ Dati in output

Noti! Struttura nota! Ignoti! Struttura nota!

Problemi e istanze

Problema: es. data una sequenza di numeri interi, trovarne il maggiore.

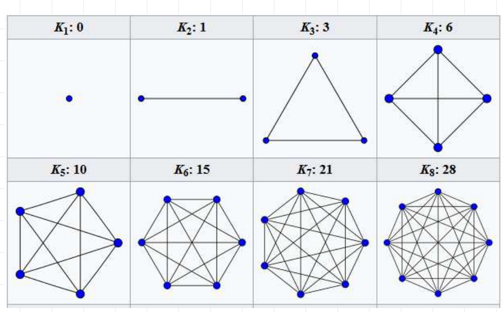
Istanza: dato (3 6 4 8 9 1), trovare il maggiore.

Istanza: dato (3 8 5 1), trovare il maggiore.

Istanza: dato (3 65 7 23 1), trovare il maggiore.

Problema: es. dato un grafo completo, trovarne un ciclo hamiltoniano.

Istanze



Algoritmi

Cos'è un algoritmo? Intuitivamente: un modo per risolvere un problema.

Dato un problema, una sua risoluzione è un processo che trasforma i dati in ingresso nei corrispondenti dati finali.

"Un algoritmo è una sequenza di azioni non ambigue che risolve un problema utilizzando un insieme di azioni elementari che possono essere eseguite da un opportuno esecutore".

Un algoritmo ***non è un programma***: "rappresentazione di un algoritmo utilizzando un linguaggio non ambiguo e direttamente comprensibile dal computer".

Algoritmi: esempio

Esempio? (*adattato da http://www.piadinaonline.com*)

1. Disporre la farina sul tavolo creando un buco nel mezzo.
2. Sciogliere lo strutto con l'acqua tiepida.
3. Impastare il sale, l'acqua e lo strutto con la farina.
4. Fare riposare l'impasto per circa 30 minuti.
5. Preparare pagnottine di circa 150 gr. ciascuna.
6. Tirare le piadine con il matterello fino a che raggiungono uno spessore di circa 0,5 cm.
7. Strofinare frequentemente il mattarello con della farina per evitare che l'impasto vi si attacchi.
8. Cuocere la piada su di una teglia in ghisa o in terracotta sotto cui deve ardere un fuoco allegro.
9. Sforacchiare entrambe le superfici della piadina con una forchetta per migliorare la cottura interna.
10. Utilizzate un coltello lungo a lama larga per rigirare in senso orario e rivoltare spesso la piadina.

Algoritmi

Proprietà degli algoritmi:

- Input da un insieme ben specificato (stringhe, numeri ..),
- Output da un insieme ben specificato,
- Non ambiguità di ogni passo di elaborazione,
- Eseguibilità di ogni istruzione in tempo finito,
- Correttezza del risultato per ogni possibile input,
- Finitezza del numero di passi di elaborazione,
- Efficacia di ogni passo di elaborazione,
- Generalità per una classe di problemi.

Algoritmi

Ogni problema può essere risolto in molti modi diversi

$\Downarrow$

Molti algoritmi di soluzione per uno stesso problema.

Quale conviene usare?

Necessario:

1. Saperli rappresentare $\rightarrow$ Pseudocodice
2. Saperli confrontare $\rightarrow$ Notazione O grande

Pseudocodice

- Un algoritmo è espresso come sequenza di azioni elementari (passi) da eseguire per risolvere il problema.
- I passi vengono rappresentati con un linguaggio astratto ed informale: lo *pseudocodice*.
- Lo pseudocodice utilizza la struttura di un linguaggio di programmazione normale, ma è inteso per uso umano e non di una macchina: omette dettagli essenziali (dichiarazioni di variabili, subroutine ecc.).
- Il linguaggio è inoltre aumentato con passi in linguaggio naturale, o con notazioni matematiche compatte.
- Non esiste nessuno standard per la sintassi dello pseudocodice, ognuno può scriverselo come vuole.

Pseudocodice

Parole chiave comuni:

- `do while ... endDo;`
- `repeat ... until;`
- `case ... endCase;`
- `if ... endif;`
- `return ...;`

- Blocchi sempre indentati! Racchiusi o fra parentesi graffe o fra parola chiave e terminatore relativo. Terminatore opzionale.
- (sfortunatamente) gli array hanno indici che iniziano da 1.
- Spesso usati verbi (inglesi) come: generate, compute, process, let, set, reset, increment, compute, calculate, add, sum, multiply, print, display, input, output, edit, test , etc.

Pseudocodice, esempi

function max(a_1, a_2, …, a_n)
max = a_1
for i = 2 **to** n
    if max < a_i **then** max = a_i

Ci può essere il ; come terminatore (ma anche no)

Assegnamenti: =, ←, :=

procedure linear_search(x,a_1, a_2, …, a_n)
i = 1
while (i <= n **and** x ≠ a_i)
    i = i + 1
if i <= n **then** **return**(i)
else **return**(-1)

Uguale: =, ==
Diverso: ≠, !=

Pseudocodice: esempio

Algorithm 2: GAP fixing heuristic

<u>function fixSolution</u> ($\bar{\mathbf{x}}, \mathbf{\Lambda}$);
Input : An infeasible GAP solution $\bar{\mathbf{x}}$ and a penalty vector $\mathbf{\Lambda}$
Output: A feasible solution or indication of failure
foreach $j \in J$ *such that* $\sum_{i \in I} x_{ij} = 0$ **do**
    $i_{min} \leftarrow i$ so that $q_{i_{min}j} = min_i q_{ij}$;
    $\bar{x}_{i_{min}j} \leftarrow 1$;
end
initialize $\bar{I}$ and $\bar{J}$;
foreach $i \in I$ **do**
    $\tilde{Q}_i = Q_i - \sum_{jin\bar{j}} q_{ij}\bar{x}_{ij}$;
    if $\tilde{Q}_i < 0$ **then return Fail**;
    Solve knapsack on $\tilde{Q}_i$ and get z_i;
    Add to $\bar{\mathbf{x}}$ the knapsack solution;
end
Prune overassigned clients in $\bar{\mathbf{x}}$ and update the corresponding z_i ;
$z_{\bar{F}} \leftarrow \sum_i z_i$;
if $\bar{\mathbf{x}}$ *is feasible* **then**
    return $\bar{\mathbf{x}}$
else
    return Fail
end

Sommatorie

Alcune sommatorie standard, da CLR e da *Fundamental Algorithms* di D.E. Knuth.

Serie costante (a e b interi)

$$\sum_{i=a}^{b} 1 = \begin{cases} (b-a+1) & se\ b \geq a \\ 0 & altrimenti \end{cases}$$

Serie aritmetica ($n \geq 0$)

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

Serie geometrica ($n \geq 0$ e $x \neq 1$)

$$\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}$$

se |x| < 1 allora $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$

Serie armonica ($n \geq 1$)

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \ln n$$

Confronto di algoritmi

Uno stesso problema può essere risolto in modi diversi

$\Downarrow$

Algoritmi diversi per uno stesso problema. Quale è meglio?

Esempio (che fanno?):

```
procedure p2(a1, a2, …, an)
min = a1
max = a1
for i = 2 to n
   if      (ai < min) then min = ai
   else if (ai > max) then max = ai
m = max - min
return m
```

```
procedure p4(a1, a2, …, an)
... una banale reimplementazione
ricorsiva di p1 ...
```

```
procedure p1(a1, a2, …, an)
m = 0
for i = 1 to n-1
   for j = i + 1 to n
      if (|ai - aj| > m)
      then m = |ai - aj|
return m
```

```
procedure p3(a1, a2, …, an)
if(n=2)
   m = |a2 - a1|
else
   m = max(p3(a2, …, an),
           |ai - a1| i=2,…,n)
return m
```

Confronto di algoritmi

Cosa rende un programma migliore di un altro?

Molti criteri possibili: leggibilità del codice, paradigmi di sviluppo, efficienza ...

Non interessano gli aspetti che derivano dal linguaggio, dalla piattaforma o dall'implementazione.

Interessa il metodo: l'algoritmo e l'utilizzo di risorse che implica.

Le risorse possono essere diverse: tempo di esecuzione, memoria richiesta, numero di CPU, banda passante per la comunicazione, energia elettrica consumata ...

Due risorse principali: tempo e spazio, spesso in trade-off. A seconda della situazione, si può voler peggiorare una per migliorare l'altra.

Fattori del tempo di CPU

Dimensione dell'input → crescita lineare tempo di lettura

Tempo di elaborazione → dipende dall'algoritmo!

Velocità di accesso alla memoria → varia di un fattore

Velocità CPU → varia di un fattore

num di processori → varia di un fattore.

Ottimizzazione del compilatore / linker → ~20%

Tempo CPU per un programma

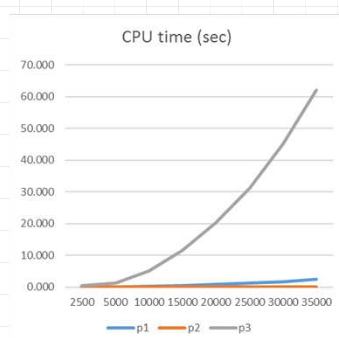
Latency Comparison Numbers					
L1 cache reference	0.5	ns			
Mutex lock/unlock	25	ns			
Main memory reference	100	ns			
Send 1K bytes over 1 Gbps network	10,000	ns	10 µs		
Read 4K randomly from SSD	150,000	ns	150 µs		~1GB/sec SSD
Read 1 MB sequentially from memory	250,000	ns	250 µs		
Round trip within same datacenter	500,000	ns	500 µs		
Read 1 MB sequentially from SSD	1,000,000	ns	1,000 µs	1 ms	~1GB/sec SSD, 4X memory
Disk seek	10,000,000	ns	10,000 µs	10 ms	20x datacenter roundtrip
Read 1 MB sequentially from disk	20,000,000	ns	20,000 µs	20 ms	80x memory, 20X SSD
Send packet CA->Netherlands->CA	150,000,000	ns	150,000 µs	150 ms	

Notes
1 ns = 10^-9 seconds
1 µs = 10^-6 seconds = 1,000 ns
1 ms = 10^-3 seconds = 1,000 µs = 1,000,000 ns

Credit ------ Jeff Dean: http://research.google.com/people/jeff/

Efficienze

	p1	p2	p3
2500	0.021	0.001	0.342
5000	0.067	0.001	1.279
10000	0.205	0.001	4.970
15000	0.438	0.002	11.457
20000	0.788	0.002	20.152
25000	1.238	0.002	31.166
30000	1.689	0.002	45.096
35000	2.373	0.002	62.033



Indipendenza dalla tecnologia

Il confronto su istanze benchmark basato sul tempo effettivo di esecuzione non è molto significativo.

Il tempo dipende da molti fattori: linguaggio di programmazione utilizzato, compilatore, linker, sistema operativo, piattaforma di calcolo, carico del processore (ora del giorno), ...

Ad esempio, serve molto più tempo su computer più vecchi.

Serve una metrica più astratta per caratterizzare gli algoritmi rispetto al tempo di esecuzione, indipendente dalla caratteristiche della piattaforma utilizzata per l'esecuzione.

Passi e dimensioni

- Un algoritmo è una sequenza di passaggi necessari per risolvere un problema: il tempo di esecuzione di un algoritmo può essere espresso come il numero di passi necessari per risolvere il problema.
- Questa astrazione caratterizza l'efficienza di un algoritmo pur rimanendo indipendente da computer o programma
- I passi più costosi del calcolo in p2 sono le assegnazione di variabili. Contandole si ha un'approssimazione del tempo di esecuzione: tre istruzioni di assegnazione (min,max,m) che vengono eseguite solo una volta, e un ciclo che esegue un assegnamento $\mathrm{n}\text{-}1$ volte.
- Possiamo denotare questo con funzione T, dove $\mathrm{T}(n) = 3 + n - 1$.
- Il parametro n è definito come la "dimensione del problema", così possiamo leggere la funzione come "T(n) è il tempo necessario per risolvere un problema di dimensione n"

O grande

La dimensione del problema per p2 è il numero di interi in input: una istanza con 100000 interi è più grande di una con 1000.

E' ovvio che il tempo di esecuzione cresce con la dimensione dell'istanza: Quanto?

Non interessa il numero esatto di istruzioni, si considera solo il termine dominante. Così si ottiene una approssimazione di T(n).

L'*ordine di grandezza* di T(n) dipende solo dal termine che cresce di più al crescere di n. "Ordine di grandezza" è lungo, si scrive O grande.

Si scrive $\mathrm{O}(f(n))$, dove $f(n)$ è il termine dominante dell'originale $\mathrm{T}(n)$. Questa è la "notazione O grande", una approssimazione del numero di passaggi in un calcolo.

In p2, $\mathrm{T}(n)=3+n\text{-}1$. Al crescere di n, la costante 3 diventa insignificante. Una buona approssimazione di T(n) quindi è O(n).

O grande, esempio

Come esempio, supponiamo $\mathrm{T}(n)=5\mathrm{n}^3+27\mathrm{n}^2+10000.$

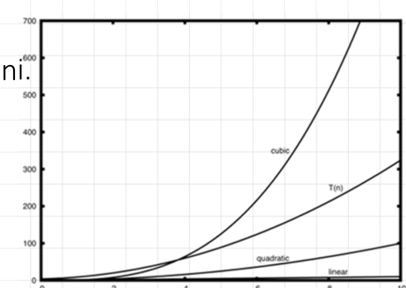
Con n piccolo (< 4), la costante10000 è dominante e $\mathrm{x}^2 > \mathrm{x}^3$.

Col crescere di n, x^3 diventa molto più grande degli altri termini.

Per approssimare $\mathrm{T(n)}$ per n grandi, si può tenere $5x^3$ e ignorare gli altri termini.

Anche il coefficiente 5 diventa non significativo per n grandi.

La funzione T(n) ha un ordine di grandezza $f(n)=n^3$: $\mathrm{T}(n)$ è $\mathrm{O}(n^3)$.



Complessitá

Un algoritmo $O(n^3)$ non può essere usato per istanze molto grandi, mentre un algoritmo $O(n)$ è utilizzabile.

Quello che interessa è il *tasso di crescita* delle funzioni di complessità.

La funzione di crescita della complessità in spazio e tempo, al crescere della dimensione dell'istanza n è una misura utilizzabile come base per il confronto di algoritmi.

La crescita delle funzioni

La crescita delle funzioni può essere descritta con la notazione O grande.

Definizione: Siano f e g due funzioni da R in R.

Diciamo che $f(x)$ è $\mathrm{O}(g(x))$ se esistono due costanti C e k tali per cui

$f(x) \leq \mathrm{C} \cdot g(x)$ quando $x > \mathrm{k}$.

Quando si analizza la crescita di funzioni di complessità, $f(x)$ e $g(x)$ si assumono sempre positive.

Quando si vuole dimostrare che $f(x)$ è $\mathrm{O}(g(x))$, è sufficiente trovare una coppia (C, k) per cui vale la relazione (ce ne possono essere infinite).

La crescita delle funzioni

L'idea della notazione O grande è definire un limite superiore alla crescita della funzione $f(x)$ per x grandi.

Questo limite è specificato dalla funzione $g(x)$, che è scelta molto più semplice di $f(x)$.

È possibile utilizzare la costante C in $f(x) \leq C \cdot g(x)$ quando $x > k$, perché **C** non cresce con $\boldsymbol{x}$.

Interessano solo x grandi, quindi non importa se $f(x) > C \cdot g(x)$ per $x \leq k$.

La crescita delle funzioni

e

Esempio:

Dimostrare che $f(x) = x^2 + 2x + 1$ è $O(x^2)$.

Per $x > 1$ abbiamo:

$x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 \Rightarrow x^2 + 2x + 1 \leq 4x^2$

Quindi, per $C = 4$ e $k = 1$: $f(x) \leq Cx^2$ quando $x > k$.

$\Rightarrow f(x)$ è $O(x^2)$.

La crescita delle funzioni

Domanda: se $f(x)$ è $O(x^2)$, è anche $O(x^3)$?

Si. x^3 cresce più di x^2, quindi x^3 cresce anche più di $f(x)$.

Dobbiamo sempre trovare la più piccola funzione semplice $g(x)$ per cui $f(x)$ è $O(g(x))$.

Casi ottimo, medio, pessimo

Spesso il tempo di esecuzione dipende dai valori letti in input, non dalla dimensione dell'input (lo vedremo con ordinamenti, knapsack, ...). Istanze grandi uguali possono avere tempi di esecuzione molto diversi.

Per questi algoritmi dobbiamo definire una caratterizzazione del *caso pessimo*, *caso ottimo*, e *caso medio*.

- Il caso pessimo lo si ha per input che rendono massimo il tempo di esecuzione.
- Il caso ottimo lo si ha per input che rendono minimo il tempo di esecuzione.
- Il caso medio lo si ha con riferimento all'intero universo di input possibili, lo si può identificare con metodo statistici, spesso complessi.

Esempio: ricerca sequenziale

Restituisce l'indice della prima occorrenza del valore val nell'array v[]. Ritorna −1 se il valore non è presente.

```
procedure RicSequenziale (val, v[])
for(i=1 ... v.length)
   if(v[i]==val) then return i
return -1
```

Nel caso ottimo l'elemento è all'inizio della lista, e viene trovato alla prima iterazione. Quindi Tbest(n) =O(1)

Nel caso pessimo l'elemento non è presente nella lista (oppure è presente nell'ultima posizione), quindi si itera su tutti gli elementi. Quindi Tworst(n) = Θ(n)

Nel caso medio →

Esempio: ricerca sequenziale

Caso medio

Media su tutti i possibili casi ⟹ Analisi statistica.

Non abbiamo informazioni sulla probabilità con cui si presentano i valori nella lista, dobbiamo fare delle ipotesi semplificative.

- *Assumiamo* che l'elemento sia sempre presente
- *Assumiamo* che la probabilità p_i che l'elemento cercato si trovi in posizione i ($i = 1, 2, \dots n$) sia $p_i = \frac{1}{n}$ per ogni i.

Il tempo T(i) necessario per individuare l'elemento nella posizione i-esima è T(i) =i. Quindi possiamo concludere che:

$$T_{\{avg\}}(n) = \sum_{i=1}^{n} p_i T(i) = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n(n+1)}{2} = \Theta(n)$$

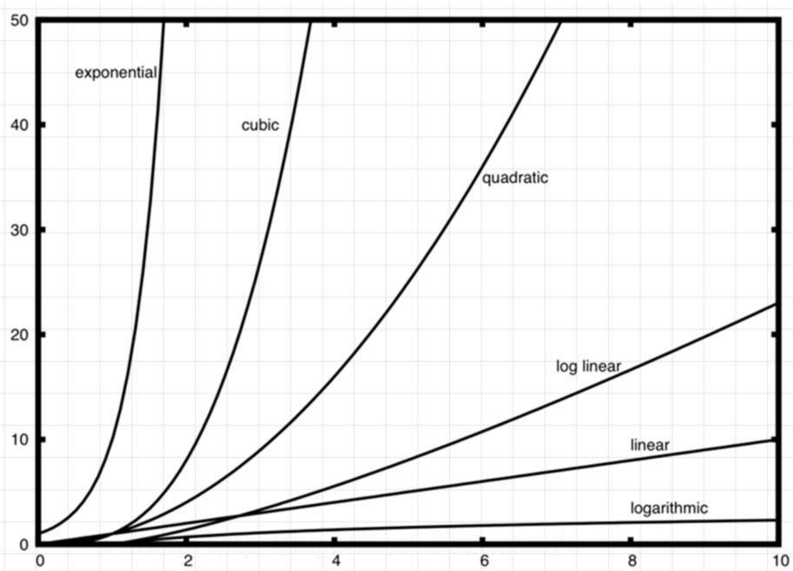
Funzioni di riferimento

Ci sono alcune funzioni di crescita semplici comunemente utilizzate nello studio degli algoritmi.

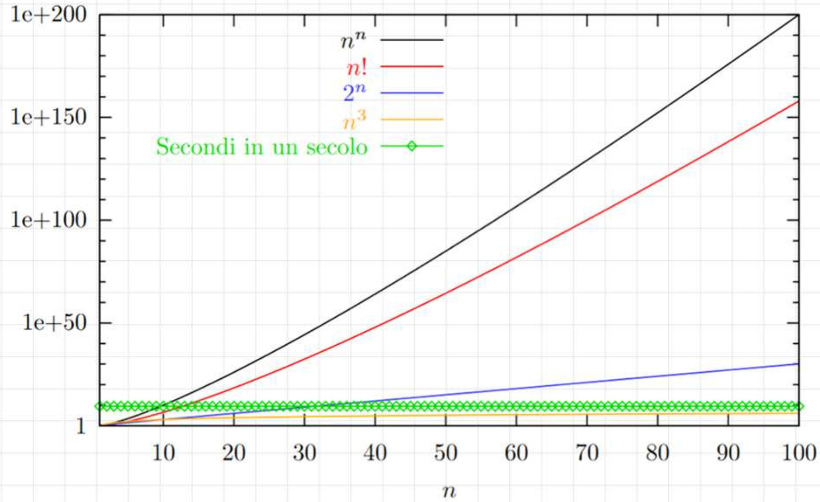
In ordine di ordine di grandezza crescente:

$f(n)$	nome
1	Costante
$log\ n$	Logaritmica
n	Lineare
$n\ log\ n$	Log lineare
n^2	Quadratica
n^3	Cubica
2^n	Esponenziale

Crescita funzioni



Crescita funzioni



Nota: scala *y* logaritmica!

Intuitivamente

$O(1)$ — tempo costante: dato un input di dimensione n, serve sempre un numero fisso di passi per risolvere il problema.

$O(\log n)$ — tempo logaritmico: dato un input di dimensione n, il numero di dati da considerare per risolvere il problema cala di un fattore ad ogni passo (es. si dimezza ad ogni passo)

$O(n)$ — tempo lineare: dato un input di dimensione n, il numero di passi richiesti è proporzionale al numero di dati (es. ciclo for)

$O(n^2)$ — tempo quadratico: dato un input di dimensione n, il numero di passi cresce col quadrato di n (es. due for annidati)

$O(C^n)$ — tempo esponenziale: dato un input di dimensione n, il numero di dati da considerare per risolvere il problema cresce di un fattore ad ogni passo (es. si raddoppia ad ogni passo). Tipico di situazioni in cui si deve considerare ogni combinazione o permutazione dei dati.

Praticalitá

O(1) — tempo costante: impossibile andare meglio.

O(log n) — tempo logaritmico: tempo delle strutture dati più efficienti per fare una singola operazione (es. inserimento in una sequenza ordinata)

O(n) — tempo lineare: il tempo minimo possibile per un algoritmo (salvo eccezioni): serve O(n) per leggere l'input.

O(n log n) — tempo loglineare: tempo di un ordinamento efficiente. Quasi tutti gli algoritmi richiedono un ordinamento di qualcosa.

$O(n^k)$ — tempo polinomiale: accettabile o quando k è piccolo o quando la dimensione dell'istanza non è troppo grande (es. n<1000)

$O(C^n)$ — tempo esponenziale: Accettabile quando: 1) l'istanza è molto piccola (es. n<50), 2) si sa che il tempo pessimo lo si avrà molto raramente

$O(n!)$, $O(n^n)$ — tempo più che esponenziale: Accettabile solo per istanze molto piccole (n<20).

E c'è anche di peggio! V. funzione di Ackermann sul testo.

Crescita dei tempi di CPU

n	O(1)	$O(\log_2 n)$	$O(n)$	$O(n \log_2 n)$	$O(n^2)$
10^2	1 μsec	1 μsec	1 μsec	1 μsec	1 μsec
10^3	1 μsec	1.5 μsec	10 μsec	15 μsec	100 μsec
10^4	1 μsec	2 μsec	100 μsec	200 μsec	10 msec
10^5	1 μsec	2.5 μsec	1 msec	2.5 msec	1 sec
10^6	1 μsec	3 μsec	10 msec	30 msec	1.7 min
10^7	1 μsec	3.5 μsec	100 msec	350 msec	2.8 hr
10^8	1 μsec	4 μsec	1 sec	4 sec	11.7 d

n	$O(n^2)$	$O(2^n)$
100	1 μsec	1 μsec
110	1.2 μsec	1 msec
120	1.4 μsec	1 sec
130	1.7 μsec	18 min
140	2.0 μsec	13 d
150	2.3 μsec	37 yr
160	2.6 μsec	37,000 yr

complessitá → massimo n

http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=complexity1

complexity	maximum N
$\Theta(N)$	100 000 000
$\Theta(N \log N)$	40 000 000
$\Theta(N^2)$	10 000
$\Theta(N^3)$	500
$\Theta(N^4)$	90
$\Theta(2^N)$	20
$\Theta(N!)$	11

Table 3. Approximate maximum problem size solvable in 8 seconds.

La crescita delle funzioni

Un problema che può essere risolto con complessità polinomiale nel caso pessimo è detto trattabile.

Problemi con complessità maggiore sono detti intrattabili.

Problemi per cui non si conosce nessun algoritmo di soluzione sono detti insolubili.

Approfondiremo alla fine del corso.

Alcune regole per O grande

Per qualunque polinomio $f(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_0$, dove a_0, a_1, ..., a_n sono numeri reali, $f(x)$ è $O(x^n)$.

Se $f_1(x)$ è $O(g(x))$ e $f_2(x)$ è $O(g(x))$, allora

$(f_1 + f_2)(x)$ è $O(g(x))$

Se $f_1(x)$ è $O(g_1(x))$ e $f_2(x)$ è $O(g_2(x))$, allora

$(f_1 + f_2)(x)$ è $O(max\ (g_1(x), g_2(x)))$

$(f_1 f_2)(x)$ è $O(g_1(x)\ g_2(x))$.

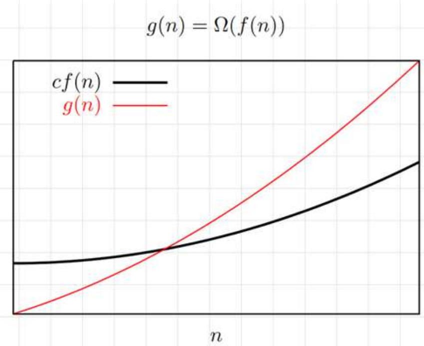
Ω grande

Limite inferiore asintotico

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

se esistono due *costanti* c e n_0, *t.c.*

$cg(n) \leq f(n)$ per $n \geq n_0$



Usato per

–tempo di esecuzione nel *caso ottimo;*

–limiti inferiori di complessità;

Esempio: il limite inferiore per la ricerca in array non ordinati è $\Omega(n)$.

Nota: si scrive $g(n) = \Omega(f(n))$ per indicare $g(n) \in \Omega(f(n))$

Ω grande

Sia $g(n) = n^3+2n^2$ e $f(n) = n^2$, dimostriamo che $g(n) = \Omega(f(n))$.

Dobbiamo trovare due costanti $c>0$, $n_0\geq 0$

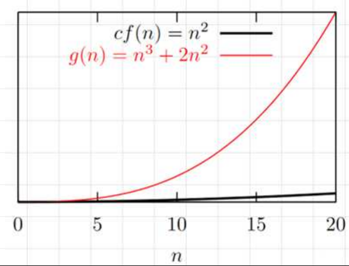
tali che per ogni $n\geq n_0$ sia $g(n)\geq cf(n)$, ossia: $n^3+2n^2\geq cn^2$ (*)

$$c \leq \frac{n^3 + 2n^2}{n^2} = n + 2$$

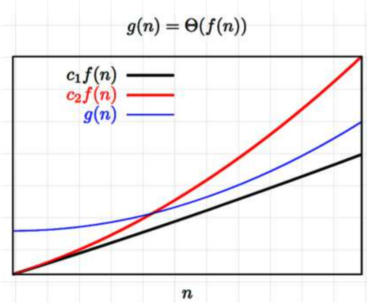
se ad esempio scegliamo

$n_0=0$ e c=1, si ha che

la relazione (*) è verificata.



Θ grande



Approssimazione *stretta*.

f(n) = Θ(g(n))

se esistono c_1, c_2, e n_0, *t.c.*
$c_1 g(n) \leq f(n) \leq c_2 g(n)$ per $n \geq n_0$

f(n) = Θ(g(n)) se e solo se *f(n)*=O*(g(n))* e *f(n)*=Ω(*g*(*n*))

O*(f(n))* spesso usato erroneamente al posto di Θ*(f(n))*

Nota: si scrive spesso $g(n) = \Theta(f(n))$ per indicare $g(n) \in \Theta(f(n))$.

"o piccolo" e "ω piccolo"

$f(n) = o(g(n))$

analogo stretto di *O* grande

- Per ogni *c*, esiste n_0, t.c. $f(n) < cg(n)$ per $n \geq n_0$
- Usato per confrontare tempi di esecuzione. Se $f(n)=o(g(n))$ diciamo che $g(n)$ domina $f(n)$.

$f(n)= \omega(g(n))$

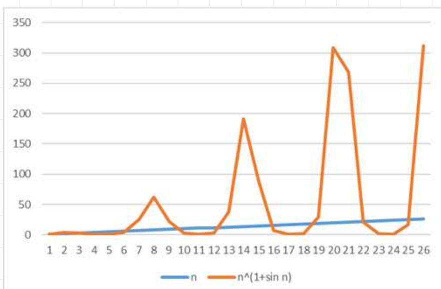
analogo stretto di Ω grande.

Non tutte le funzioni sono confrontabili

$n \quad \Longleftrightarrow \quad n^{1+sen(n)}$

???

$1+sen(n)$ oscilla tra 0 e 2



Esempi

M = numero grande, es: *10000000*;

ε = numero piccolo, es: *0,0000001*;

$$
\begin{aligned}
Mn &= \Theta(n\varepsilon) \\
log(n) &= o(n) \\
[log(n)]^M &= o(n^{\varepsilon}) \\
n^M &= o(2^{n\varepsilon}) \\
2^{nM} &= o(n!) \\
Mn! &= o(n^n)
\end{aligned}
$$

Limiti e notazione asintotica

$\lim_{n\to\infty} \frac{f(n)}{g(n)} \to c$ allora $f(n) = \Theta(g(n))$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} \to 0$ allora $f(n) = o(g(n))$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} \to \infty$ allora $f(n) = \omega(g(n))$