Analisi e Progettazione di Algoritmi

a.a. 2019-20

Tre moduli

- Modulo 1 (Zucca): tecniche per progettazione e analisi, grafi e NP-completezza
- Modulo 2 (Verri): algoritmi randomizzati
- Modulo 3 (Magillo): strutture dati e ordinamenti (per terzo anno)
- disponibili note
- modalità esame: test preliminare + orale

Primo modulo: contenuti

- TECNICHE di analisi di algoritmi (correttezza, complessità)
- TECNICHE di progettazione di algoritmi (es. programmazione dinamica)
- Algoritmi su GRAFI
- Teoria della NP-COMPLETEZZA (problemi "intrattabili", vedi calcolabilità in TAC)

Cosa dovreste imparare

- Le tecniche di progettazione e di analisi, applicate ad alcuni algoritmi che vedremo
- Per “estrapolazione”: saper progettare e analizzare algoritmi “nuovi”

TECNICHE di analisi
(in parte ripasso da ASD)

- Problema algoritmico e algoritmo

- Correttezza degli algoritmi: provare che un certo algoritmo risolve un dato problema (semi-formale)

- Efficienza degli algoritmi: analisi del tempo di calcolo e dello spazio di memoria impiegati dall'algoritmo

Introduzione informale

per rinfrescare la memoria

Problema algoritmico (computazionale)

- Massimo di una sequenza di interi: data una sequenza non vuota di numeri interi, trovare il massimo e l'indice di tale massimo
- Ordinamento: dato una sequenza di elementi su cui è definita una relazione d'ordine (totale), trovare una permutazione di tale sequenza che sia ordinata
- Ricerca in una sequenza ordinata: data una sequenza ordinata e un elemento, determinare se questo compare nella sequenza o no, e l'indice del posto in cui compare
- Cammino minimo: data una mappa stradale che riporti località, strade e lunghezze dei percorsi, e data una coppia di località (raggiungibili l'una dall'altra), trovare un percorso di lunghezza minima fra di esse
- Distanza di editing: date due stringhe, trovare la più corta sequenza di operazioni (di inserimento, cancellazione e modifica) che trasforma una stringa nell'altra
- ...

Problema algoritmico (computazionale)

Più formalmente:

- Relazione $\mathcal{P}$ su I × S dove
- I insieme degli input, o istanze del problema
- S insieme delle soluzioni
- Per ogni istanza i dell'input ho la soluzione o l'insieme delle possibili soluzioni s che si vogliono ottenere

L’enunciazione di un problema NON specifica COME ottenere la soluzione dall’input

COME ottenere la soluzione dall’input costituisce un algoritmo

Che cos’è un algoritmo ?

(descrizione precisa e non ambigua di) un procedimento di calcolo, che può quindi essere eseguito dall'uomo “meccanicamente” oppure da una macchina

Algoritmo $\mathcal{A}$ che risolve $\mathcal{P}$ = un algoritmo che fornisce, per ogni istanza del problema, un output che costituisce una soluzione

Un algoritmo può essere un programma in un linguaggio, una macchina di Turing (corso TAC), una descrizione “a parole” non ambigua (per es. algoritmo per divisione in colonna, indicazioni per trovare strada, etc.)

In questo corso non ci interessa fissare un tipo di descrizione

- “Eseguire” un algoritmo è un’attività meccanica

- Trovare (inventare) un algoritmo per risolvere (in maniera efficiente) un problema è un’attività non meccanica e non meccanizzabile – richiede “intelligenza”

Etimologia

– la parola *algoritmo* deriva dal nome del matematico uzbeko-persiano al-Khwarismi الـــخـوارزمـــي (vissuto intorno all'anno 800), il cui libro "Calcoli con i numerali indiani" descriveva alcuni algoritmi di calcolo per le operazioni aritmetiche che ancora oggi studiamo nella scuola elementare

الخوارزمي

Quando sono nati gli algoritmi ?

ben prima della nascita dei calcolatori:

- uno dei più antichi è l'algoritmo di Euclide per il MCD

- un altro è il crivello di Eratostene per la generazione dei numeri primi

L’algoritmica moderna

L’*algoritmica*, ovvero la scienza degli algoritmi, è fin dagli inizi dell’informatica una delle sue discipline centrali, ancora oggi attivo campo di ricerca

Donald Knuth, padre dell'algoritmica moderna, autore dell'opera in più volumi *The Art of Computer Programming*

Correttezza di un algoritmo
(rispetto a un problema)

- A: I $\rightarrow$ S parziale
- Per ogni input i, A(i) è una soluzione di P
- Due tecniche fondamentali:
- Induzione, per algoritmi ricorsivi
- Invarianti di ciclo, per algoritmi iterativi

Efficienza di un algoritmo

uno stesso problema può essere risolto da più algoritmi, che possono richiedere un numero di operazioni elementari e uno spazio di memoria molto diversi

domande fondamentali:

• di quanto tempo e spazio di memoria ha bisogno un algoritmo per produrre la risposta per un input di una data dimensione?

• dato un problema, si può trovare un algoritmo migliore di quelli esistenti per risolvere quel problema?

nozione formale di complessità temporale e spaziale di un algoritmo

Algoritmi e strutture dati

efficienza o addirittura definibilità di un algoritmo spesso legata al modo in cui i dati sono rappresentati
Esempio:

algoritmo di ricerca binaria
sensato – molto più efficiente della ricerca sequenziale – solo se la sequenza è realizzata per mezzo di un array (o di un albero di ricerca), NON se la sequenza è realizzata come una lista

lo studio degli algoritmi è quindi inseparabile dallo studio delle strutture dati

COMPLESSITÀ

classificare gli algoritmi a seconda della loro efficienza

avere modi precisi per analizzarli ⇓

misure di efficienza:

- spazio di memoria necessario per l’esecuzione
- tempo di calcolo

siamo interessati a determinare come lo spazio utilizzato e il tempo di calcolo variano al variare della dimensione dell’input

Andamento asintotico

Due proposte per investire uno zecchino

Tree Bank aggiunge al versamento la somma di cinque zecchini l’anno

i Buoni del Tesoro fruttano il 20% ogni anno”

Facciamo i conti:

	Versamento	1anno	2anni	3anni	. . .	10 anni
Tree Bank	1	6	11	16	. . .	51
Buoni del Tesoro	1	1,2	1,44	1,73	. . .	6,19

equazioni che regolano la crescita degli zecchini:

$1 + 5k$, se si investe in Tree Bank
$(1{,}20)^k$ se si investe in Buoni del Tesoro

k = numero di anni

Sembra migliore il primo investimento

Abbiamo scelto bene?

	10 anni	20 anni	25 anni	30 anni	40 anni	50 anni
Tree Bank	51	101	126	151	201	251
Buoni del Tesoro	6,19	38,34	95,40	237,37	1469,77	9100,44

la legge esponenziale finisce inesorabilmente per vincere

quando il suo valore supera quello di un’altra, se ne stacca con rapidità strabiliante

Tutto però è relativo: se l’investimento è per meno di 25/26 anni conviene la Tree Bank!

Classificazione delle funzioni rispetto all’andamento asintotico

	10	100	1.000	10.000	
log n	3	6	9	13	**logaritmica**
$n^{1/2}$	3	10	31	100	**radice quadr.**
n	10	100	1.000	10.000	**lineare**
n log n	30	600	9.000	130.000	**pseudolineare**
n^2	100	10.000	10^6	10^8	**quadratica**
n^3	1.000	10^6	10^9	10^{12}	**cubica**
2^n	1.024	10^{30}	10^{300}	10^{3000}	**esponenziale**

Nota: l'universo contiene circa 10^{80} particelle elementari

Quali sono più simili fra loro?

n^{1000} n^2 2^n

Le prime due!

$$\underbrace{n^{1000} \qquad\qquad n^{2}}_{\text{polinomiali}} \qquad\qquad 2^{n}$$

Quali sono più simili fra loro?

Polinomiali

$1000n^2$ $3n^2$ $2n^3$

Le prime due!

Polinomiali

$$\underbrace{1000n^2 \qquad 3n^2}_{\text{Quadratiche}} \qquad\qquad 2n^3$$

Esempio: funzioni quadratiche

- n^2
- $0.001\ n^2$
- $1000\ n^2$
- $5n^2 + 3000n + 2\log n$

Scriviamo $\Theta(n^2)$

Logaritmica << Polinomiale

$$(\log n)^{1000} \ll n^{0.001}$$

per n sufficientemente grande

Lineare << Quadratica

$$10000\, n \ll 0.0001\, n^2$$

per n sufficientemente grande

Polinomiale << Esponenziale

$$n^{1000} \ll 2^{0.001\, n}$$

per n sufficientemente grande

Cosa vuol dire crescita lineare?

$y = cx$

y è proporzionale a x

se x si raddoppia anche y si raddoppia, etc

$y = cx + d$

la proprietà precedente vale ancora approssimativamente per x sufficientemente grandi

Cosa vuol dire crescita quadratica?

$y = ax^2$ allora

y è proporzionale al *quadrato* di x

se x si moltiplica per m, y si moltiplica per m^2

anche $y = ax^2 + bx + c$ è una funzione quadratica

non è *esattamente* vero che quando x si moltiplica per m allora y si moltiplica per m^2

tuttavia al crescere di x l'influenza degli altri termini diminuisce, e la proprietà diventa approssimativamente vera

esempio: se $y = 2x^2 + 3x + 2$

per x = 1000 si ha y = 2.003.002

per x = 2000 si ha y = 8.006.002

8.006.002 / 2.003.002 = 3,997 $\approx$ 4

etc

naturalmente:

se in $y = ax^2 + bx + c$ le costanti b o c sono MOLTO più grandi di a

la proprietà diventa approssimativamente vera solo per valori *estremamente* grandi di x

Cosa vuol dire crescita logaritmica?

$y = c \log x$

quando il valore di x si moltiplica per m, cosa succede a y?
$y = c \log(m \bullet x) = c \log x + c \log m$

il valore di y NON si moltiplica, ma si *somma* con un incremento proporzionale al logaritmo di m

logaritmico = "quando x si moltiplica, y si somma”

applicazione alla complessità:
x è n (dimensione dell'input), y è T(n) (tempo di calcolo)

quando la dimensione dell'input si moltiplica, il tempo di calcolo non si moltiplica ma aumenta solo di un incremento

Cosa vuol dire crescita esponenziale?

$y = c \cdot 2^x$ o più in generale $y = c \cdot a^x$

se al valore di x si somma un incremento

$y = c \cdot 2^{(x+m)} = c\, 2^x \cdot 2^m$

il valore di y si moltiplica

esponenziale = quando x si somma, y si moltiplica

applicazione alla complessità:
x è n (dimensione dell'input), y è T(n) (tempo di calcolo)

quando la dimensione dell'input si incrementa, il tempo di calcolo non si incrementa ma si moltiplica

Esempio

tempi di esecuzione di un algoritmo per diverse dimensioni dell'input:

dimensione	tempo
5000	31
10000	140
15000	313
20000	547
25000	875
30000	1250
35000	1859
40000	2297
45000	2750

Che cosa si può dire dell'andamento asintotico del tempo di calcolo?

Esempio

tempi di esecuzione di un dato algoritmo per diverse dimensioni dell'input:

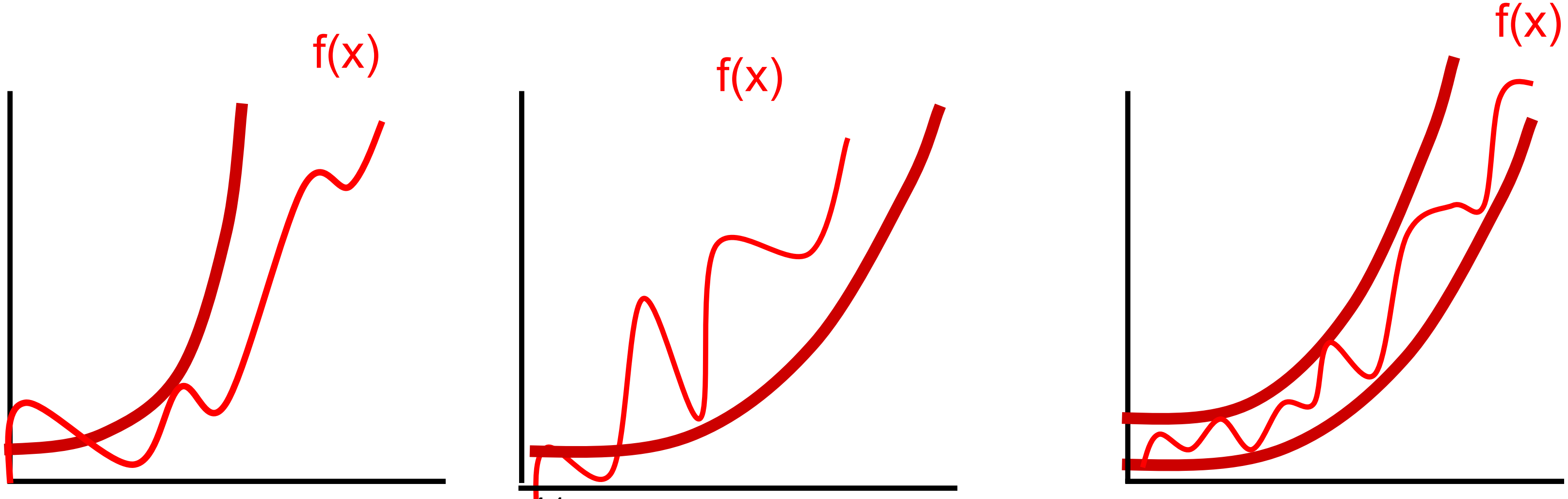
dimensione	tempo
3	7
4	15
5	31
6	63
7	127
8	255
9	511
10	1023

Che cosa si può dire dell'andamento asintotico del tempo di calcolo?

Andamento asintotico di funzioni:
definizioni di **O**, **Θ**, **Ω**

f, da un certo punto in poi,

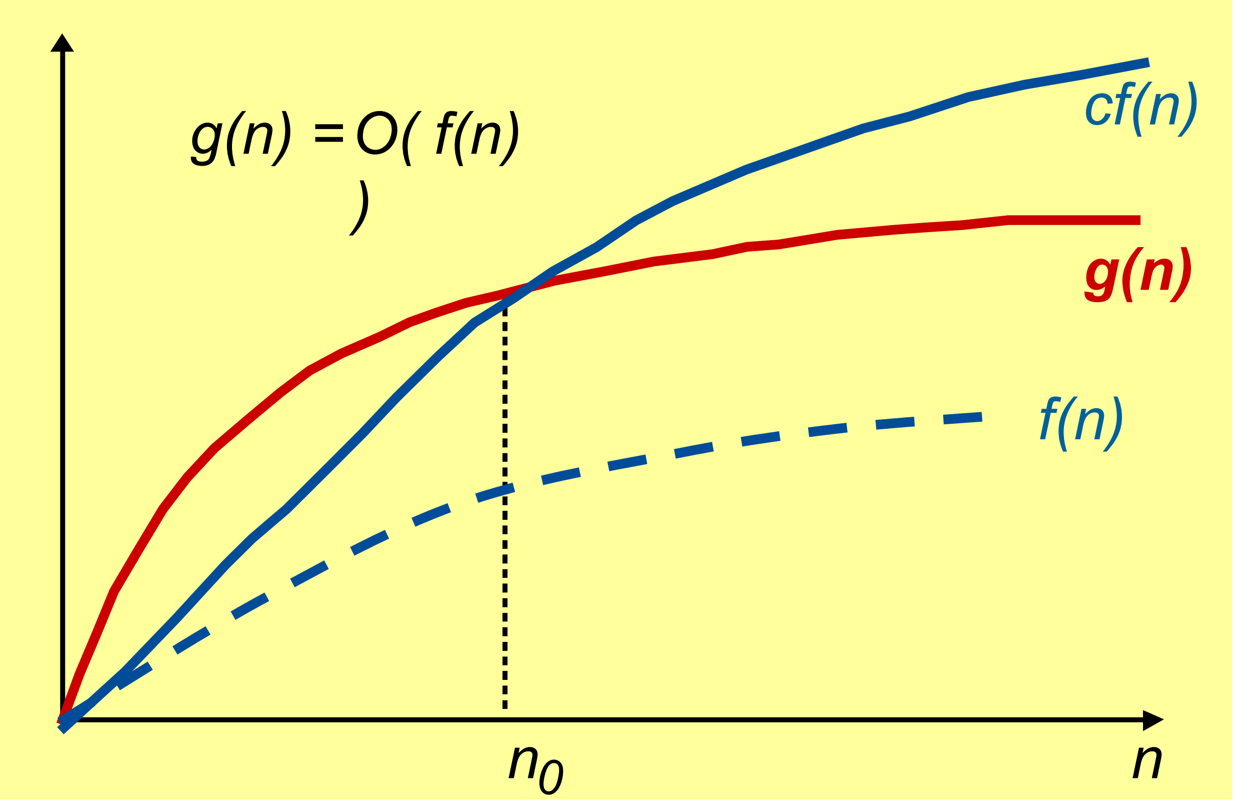
- cresce non più di una funzione di un certo genere
- cresce non meno di una funzione di un certo genere
- cresce come le funzioni di un certo genere



andamento asintotico O ("o grande")

$g(n)$ è $O(f(n))$ se esistono due costanti $c>0$ e $n_0 \geq 0$ tali che $g(n) \leq c\, f(n)$ per ogni $n \geq n_0$

g cresce **al più** come **f**
(o meglio, al più come un "multiplo" di f)



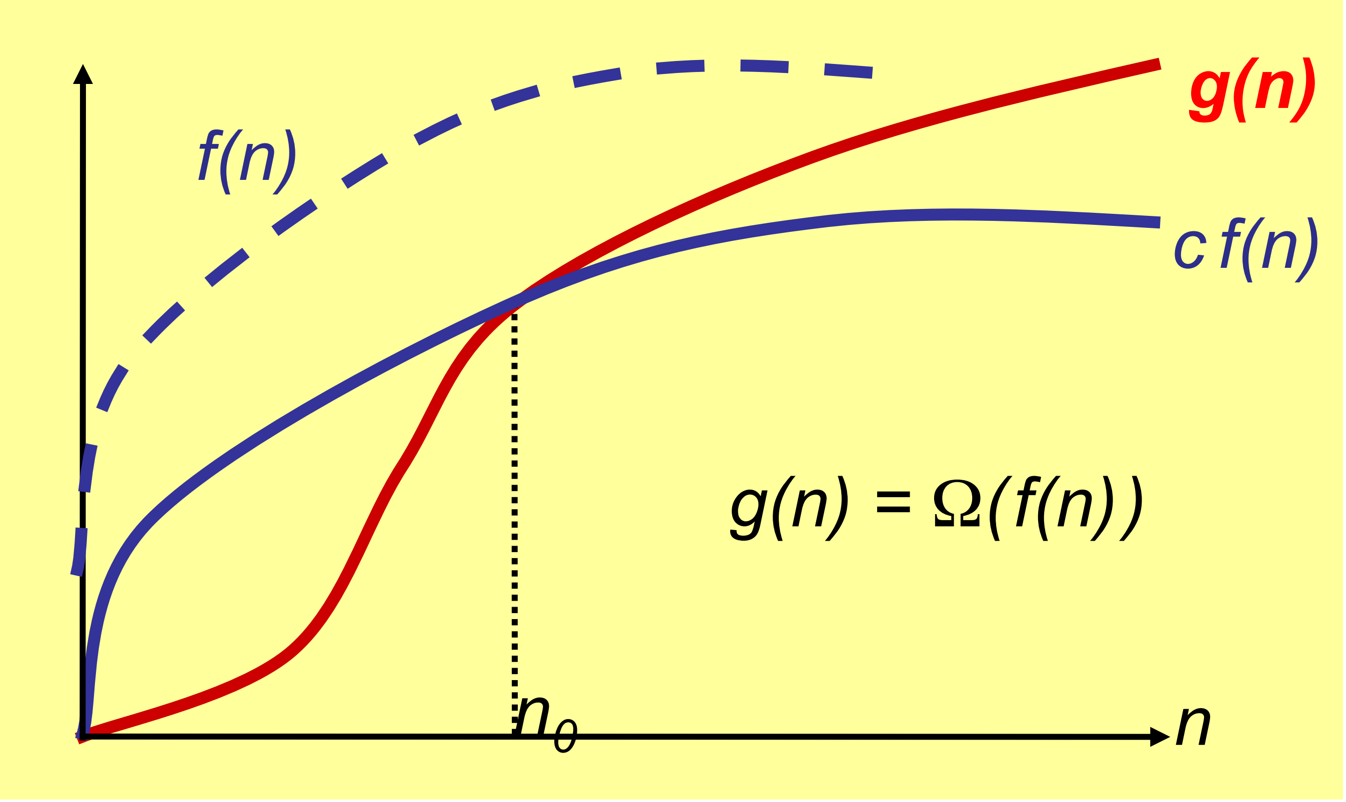
andamento asintotico Ω (“omega grande”)

g(n) è Ω(f(n)) se esistono due costanti $c>0$ e $n_0 \geq 0$ tali che

g(n) ≥ c f(n) per ogni $n \geq n_0$

g cresce **almeno** come **f**

(o meglio, almeno come un "sottomultiplo" di f)

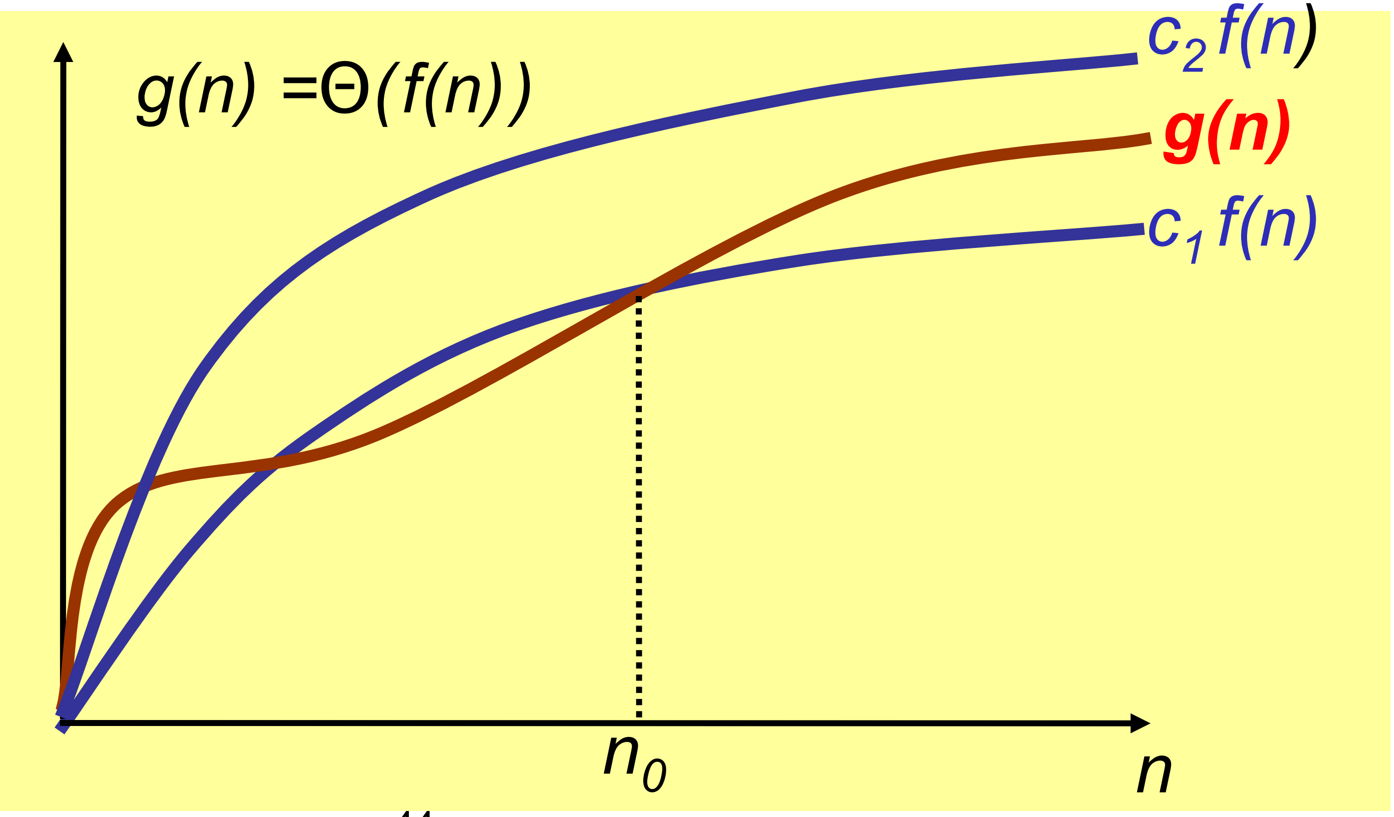


andamento asintotico Θ (theta)

$g(n)$ è $\Theta(f(n))$ se esistono tre costanti $c_1, c_2 > 0$ e $n_0 \geq 0$ tali che

$c_1 f(n) \leq g(n) \leq c_2 f(n)$ per ogni $n \geq n_0$

g cresce **esattamente come f**



Nota: abuso di linguaggio

"multiplo di f" e "sottomultiplo di f" sono un abuso di linguaggio per indicare $c\ f(n)$, c costante reale positiva

tale abuso è giustificato dalle osservazioni seguenti:

- nella definizione di O è significativo il fatto che c possa essere **>** 1, mentre nella definizione di Ω è significativo che c possa essere **<** 1
- ossia, nel primo caso ci si potrebbe limitare a considerare costanti positive intere ("multipli" di f)
- nel secondo caso ci si potrebbe limitare a considerare costanti della forma 1/h, con h positivo intero ("sottomultipli" di f)

un abuso di notazione consolidato

si usa scrivere

$g(n) = O(f(n))$ invece di **$g(n)$ è $O(f(n))$**, ecc.

abuso di notazione poiché il simbolo "**=**" in questo caso denota l'appartenenza a un insieme:

$O(f(n)) = \{h(n) \mid$ esistono $c>0$ e $n_0 \geq 0$ tali che $h(n) \leq c\, f(n)$ per ogni $n \geq n_0 \}$

analogamente per Ω e Θ

proprietà abbastanza ovvie:

<u>Transitiva</u>:

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

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

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

<u>Riflessiva</u>:

$f(n) = \Theta(f(n))$

$f(n) = O(f(n))$

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

<u>Simmetrica</u>: $f(n) = \Theta(g(n))$ $\leftrightarrow$ $g(n) = \Theta(f(n))$

<u>Simmetrica trasposta</u>: $f(n) = O(g(n))$ $\leftrightarrow$ $g(n) = \Omega(f(n))$

Comportamenti notevoli in ordine crescente:

$\Theta(1)$	costante
$\Theta(\log n)$	logaritmico
$\Theta(n)$	lineare
$\Theta(n \log n)$	pseudolineare
$\Theta(n^2)$	quadratico
$\Theta(n^3)$	cubico

trattabili

$\Theta(2^n)$	esponenziale
$\Theta(n!)$	fattoriale
$\Theta(n^n)$	(esponenziale in base n)

intrattabili

Complessità computazionale degli algoritmi

Tempo e spazio di calcolo

in generale sia il numero di passi elementari che lo spazio richiesto da un algoritmo dipendono dalla dimensione n dell'input

vogliamo valutare come tempo e spazio dipendono da n, cioè valutare l'andamento delle funzioni T(n) ed S(n)

ma ...

in generale **T(n)** ed **S(n)** ***non sono funzioni***

Caso peggiore, caso migliore, caso medio

a parità di dimensione, input diversi possono dar luogo a tempi di calcolo e spazi di occupazione molto diversi

esempio: insertion sort

- le notazioni **T(n)** ed **S(n)** sono quindi ambigue
- si distingue allora, per ogni n, fra **caso peggiore**, **caso migliore**, e **caso medio**

i input (o *istanza* del problema)

- **t(*i*)** = tempo di esecuzione dell'algoritmo per l'input ***i***
- **s(*i*)** = spazio di memoria necessario, **in aggiunta allo spazio occupato dall'input stesso**, per l'esecuzione dell'algoritmo per l'input ***i***
- $\mathbf{n}_{i}$ = dimensione dell'input ***i***, espressa in unità convenienti (per esempio, per un algoritmo che opera su array, il numero di elementi)

t(*i*) ed **s(*i*)** di solito dipendono da $\mathbf{n}_{i}$, ma NON SOLO da $\mathbf{n}_{i}$

- complessità temporale del caso peggiore:
  $\mathbf{T_{worst}(n)} = \max \{t(\boldsymbol{i}) \mid \boldsymbol{i} \text{ ha dimensione } n\}$
- complessità temporale del caso migliore:
  $\mathbf{T_{best}(n)} = \min \{t(\boldsymbol{i}) \mid \boldsymbol{i} \text{ ha dimensione } n\}$
- complessità temporale del caso medio:
  $\mathbf{T_{avg}(n)} = \text{media} \{t(\boldsymbol{i}) \mid \boldsymbol{i} \text{ ha dimensione } n\}$

per ogni **n**, i valori $\mathbf{T_{worst}(n)}$, $\mathbf{T_{best}(n)}$ e $\mathbf{T_{avg}(n)}$ sono univocamente determinati

$\mathbf{T_{worst}(n)}$, $\mathbf{T_{best}(n)}$ e $\mathbf{T_{avg}(n)}$ sono quindi vere **funzioni** (di **n**)

- le definizioni e le proprietà per la complessità ***spaziale*** sono perfettamente analoghe

Osservazioni sul caso medio

$\mathbf{T_{avg}(n)}$ = media $\{t(\boldsymbol{i}) \mid \boldsymbol{i}$ ha dimensione n$\}$

per ogni **n** si considerano tutti i possibili input di dimensione **n**: $\boldsymbol{i_1}, \ldots, \boldsymbol{i_M}$

e si fa la media aritmetica dei tempi :

$$\mathbf{T_{avg}(n) = (t(\boldsymbol{i_1}) + \ldots + t(\boldsymbol{i_M})\ )\ /\ M}$$

se i possibili casi di input si presentano con probabilità diverse $\mathbf{p_1, \ldots, p_M}$,

$$\mathbf{p_1 + p_2 + \ldots + p_M = 1}$$

la media deve essere una media pesata:

$$\mathbf{T_{avg}(n) = p_1 \cdot t(\boldsymbol{i_1}) \ldots + p_M \cdot t(\boldsymbol{i_M})}$$

Delimitazioni superiori e inferiori

nove possibili affermazioni su complessità algoritmo

$T_x(n)$ è **$O(f(n))$** o **$\Omega(f(n))$** o **$\Theta(f(n))$**, con **X = best, worst, avg**

non tutte fra loro indipendenti né ugualmente interessanti

in particolare, per ogni algoritmo si ha:

$T_{worst}(n)=O(f(n))$ **$T_{avg}(n)=O(f(n))$** **$T_{best}(n)=O(f(n))$**

se il tempo di calcolo cresce non più di $f(n)$ nel caso peggiore, ovviamente cresce non più di $f(n)$ anche nei casi migliore e medio

Simmetricamente:

$T_{best}(n)=\Omega(f(n))$ ⟹ $T_{medio}(n)=\Omega(f(n))$ ⟹ $T_{worst}(n)=\Omega(f(n))$

se il tempo di calcolo cresce almeno come $f(n)$ nel caso migliore, ovviamente cresce almeno come $f(n)$ anche nei casi medio e peggiore

Delimitazioni superiori e inferiori

sono quindi giustificate le seguenti definizioni:

- un algoritmo **ha complessità O(f(n))** se $\mathbf{T_{worst}(n)=O(f(n))}$

  f(n) è una **delimitazione superiore** del tempo di calcolo per qualsiasi input

  al crescere di n **il tempo di calcolo non cresce di più di f(n), qualunque sia l'input**

un algoritmo **ha complessità Ω(f(n))** se è $\mathbf{T_{best}(n)=\Omega(f(n))}$

f(n) è una **delimitazione inferiore** del tempo di calcolo per qualsiasi input

al crescere di n **il tempo di calcolo cresce almeno come f(n), qualunque sia l'input** un algoritmo **ha complessità $\Theta(f(n))$**

se ha complessità **$O(f(n))$** e **$\Omega(f(n))$**

f(n) è una **delimitazione sia inferiore che superiore** del tempo di calcolo per qualsiasi input

equivalentemente, diciamo anche che:

il tempo di esecuzione di un algoritmo è **$O(f(n))$** se **$T_{worst}(n)$** è **$O(f(n))$**

Esempio: insertion sort

il tempo **nel caso peggiore** è Θ**(n^2),** quindi anche **O(n^2)**
quindi, per definizione,
il tempo di esecuzione dell'algoritmo è **O(n^2)**

Il tempo **nel caso migliore** è Θ**(n),** quindi anche **Ω(n)**
quindi, per definizione,
il tempo di esecuzione dell'algoritmo è **Ω(n)**

Il tempo di esecuzione *non è* né Θ**(n^2)** né Θ**(n)**:
per ogni dimensione **n** vi sono sia input per cui il tempo è proporzionale a **n**, sia input per cui è proporzionale a **n^2**

Osservazioni metodologiche

- **caso migliore**: di solito scarsamente interessante, riguarda input molto particolari; eccezione per esempio **insertion sort**, il cui caso migliore (sequenza ordinata) è un caso significativo

- **caso medio**: significativo solo se distribuzione delle probabilità riflette situazione reale: determinazione di tale distribuzione può essere difficile; inoltre in applicazioni critiche l'efficienza nel caso medio può non essere garanzia sufficiente; eccezione per esempio **quicksort**, in cui il caso medio è quello interessante

- **caso peggiore**: complessità che si studia di solito, poiché unica che fornisce la garanzia che in ogni caso il tempo di esecuzione non sarà maggiore di un certo tempo prevedibile

Complessità di problemi

Delimitazione superiore (upper bound) di un problema

un problema ha complessità $O(f(n))$ se *esiste* un algoritmo di complessità $O(f(n))$ che lo risolve

è possibile risolverlo in un tempo che cresca non più di f(n)

Delimitazione inferiore (lower bound)

di un problema

un problema ha complessità $\Omega(f(n))$ se *tutti* i possibili algoritmi risolventi (anche quelli non ancora inventati) hanno complessità $\Omega(f(n))$

ossia se qualunque algoritmo che risolva il problema *deve* avere complessità del caso peggiore/migliore/medio $\Omega(f(n))$

è impossibile risolverlo in un tempo che cresca meno di f(n)

QUINDI:

- per trovare una delimitazione superiore **f(n)** alla complessità di un problema è sufficiente trovare **UN algoritmo** che risolva il problema in un tempo **O(f(n))**

per trovare una delimitazione inferiore **g(n)** è necessario **dimostrare** che qualunque possibile algoritmo, per risolvere il problema, deve impiegare un tempo **Ω(g(n))**

NON BASTA che tutti gli algoritmi conosciuti che risolvono il problema abbiano complessità **Ω(g(n))**!

Status storico dei problemi

un problema algoritmico, in un dato momento storico, può essere:

- **aperto** o con ***gap algoritmico***
- **chiuso**

ovviamente, un problema aperto può venire chiuso, ma non viceversa

Problema algoritmicamente chiuso

si conoscono limite superiore e inferiore coincidenti, ossia:

- esiste un algoritmo risolvente di complessità **O(f(n))**

si è dimostrato che qualunque algoritmo risolvente deve avere complessità **Ω(f(n))**, ossia non può esistere un algoritmo di complessità inferiore a **Ω(f(n))**

si è quindi dimostrato che l'algoritmo risolvente è ***ottimo***

saranno possibili solo miglioramenti marginali, per esempio per una costante additiva o moltiplicativa

Problema con gap algoritmico

(tutte le) delimitazioni inferiori e superiori
differiscono, ossia:

il miglior algoritmo risolvente noto ha complessità **O(f(n)),** per esempio **$O(n^2)$**

si è dimostrato che qualunque algoritmo risolvente deve avere complessità **Ω(g(n))**, con **g $\neq$ f**, per esempio **Ω(n)**

ossia:

- si sa risolvere il problema in un tempo **f(n),** per esempio **n^2**
- si sa che non lo si può risolvere in un tempo migliore di **g(n),** dove **g(n)** "cresce meno" di **f(n)**, per esempio **n**

Esempio

Pippo conosce come algoritmi di ordinamento solo selection sort e insertion sort

poiché entrambi sono quadratici nel caso peggiore, Pippo può affermare che:

la complessità del problema dell'ordinamento ha limite superiore **$O(n^2)$**

un algoritmo di ordinamento ha complessità almeno lineare in quanto deve esaminare tutti gli elementi

Pippo può quindi affermare che:

la complessità del problema dell'ordinamento ha un limite inferiore **$\Omega(n)$**

per Pippo il problema dell'ordinamento ha un gap algoritmico

Come chiudere il gap?

dal di sopra: si trova un algoritmo migliore, **abbassando così il limite superiore**

se si trova un algoritmo di complessità coincidente con il limite inferiore, tale algoritmo è ottimo e il problema è chiuso

dal di sotto: si riesce a **dimostrare un limite inferiore più alto**

se questo coincide con la complessità dell'algoritmo migliore esistente, si è dimostrato che l'algoritmo è ottimo e il problema è chiuso

i due modi non sono mutuamente esclusivi, può succedere che:

si riesca a trovare un algoritmo migliore,

contemporaneamente si riesca a dimostrare un limite inferiore più alto

Esempio

Pippo può chiudere il gap per il problema dell'ordinamento spostando entrambi i limiti:

scopre un algoritmo migliore, come mergesort, che ha complessità $O(n \log n)$; può quindi affermare che:
il problema ha limite superiore $O(n \log n)$

dimostra che il problema, a certe condizioni (ordinamento per soli confronti), richiede un numero $\Omega(n \log n)$ di passi; può quindi affermare che:
il problema ha nel caso peggiore limite inferiore $\Omega(n \log n)$

il problema dell'ordinamento per confronti è algoritmicamente chiuso

Un (importante) gap aperto

problema della soddisfacibilità di una formula booleana:

- esiste un algoritmo di *complessità esponenziale* (l'algoritmo banale che prova tutte le possibili combinazioni di valori di verità)
- è facile dimostrare che qualunque algoritmo risolvente deve avere *complessità almeno polinomiale*

ma:

- nessuno è riuscito a trovare un algoritmo risolvente polinomiale
- nessuno è riuscito a dimostrare che un tale algoritmo (cioè polinomiale) non può esistere, anche se si ha il fondato sospetto che sia così

- vi sono parecchi problemi come questo, per i quali gli unici algoritmi risolventi sono esponenziali, e si sospetta fortemente – ma non si è dimostrato – che non esistano algoritmi migliori

un altro esempio famoso è quello del commesso viaggiatore

- si tratta in generale di problemi per i quali, se si possiede la soluzione (per esempio un'assegnazione di valori di verità che rende soddisfacibile la formula booleana), verificare che essa è corretta è facile, cioè può essere fatto in modo efficiente

- ci torneremo a fine modulo: classe NP e problemi NP-completi

Problemi algoritmici “intrinsecamente difficili”

- per altri problemi algoritmici gli unici algoritmi risolventi sono esponenziali, e *si è dimostrato* che non possono esistere algoritmi migliori

- esempio: le torri di Hanoi

- si tratta in generale di problemi per i quali anche solo verificare che la soluzione è corretta richiede un tempo esponenziale

Problemi algoritmici non risolubili

non tutti i problemi algoritmici sono risolubili: per alcuni si può dimostrare che non esiste un algoritmo che risolve il problema (teoria calcolabilità vista a TAC)

Esempi

dati due programmi, determinare se essi sono equivalenti, cioè se per qualunque input producono lo stesso output

dato un programma e un input legale per tale programma, determinare se il programma, per quell'input, termina

Riassunto: generi di problemi algoritmici

- trattabili chiusi: risolti da algoritmo efficiente (logaritmico, lineare, pseudo-lineare) o almeno polinomiale, e si è dimostrato che non esistono algoritmi asintoticamente migliori

  esempio: il problema dell'ordinamento
- trattabili con gap algoritmico: risolti da algoritmo efficiente o almeno polinomiale, che però non si sa se sia quello asintoticamente migliore

  esempio: prodotto di matrici
- "presumibilmente" intrattabili: risolti da algoritmo esponenziale, e si sospetta – ma non si è dimostrato – che non esistano algoritmi migliori

  esempi: soddisfacibilità booleana, problema del commesso viaggiatore
- dimostrabilmente intrattabili: gli unici algoritmi risolventi sono esponenziali, e si è dimostrato che non possono esistere algoritmi migliori

  esempi: torri di Hanoi
- dimostrabilmente insolubili: si è dimostrato che non possono esistere algoritmi risolventi

  esempi: problema della terminazione, problema dell'equivalenza fra programmi