Algoritmi e strutture dati

Terminazione, correttezza, e complessitá

Menú di questa lezione

In questa lezione parleremo di come si analizza un algoritmo e di come si studia la sua complessità, distinguendo tra i casi iterativo e ricorsivo. Introdurremo anche i primi rudimenti di notazione asintotica.

Analizzare gli algoritmi

Ci sono quattro caratteristiche fondamentali che ci interessano di un algoritmo: **correttezza**; **completezza**; **terminazione**; e **complessità**. Affermare che un algoritmo è **corretto** significa affermare che esso restituisce sempre una risposta corretta. Affermare che è **completo** significa affermare che ogni risposta corretta è, prima o poi, effettivamente restituita. Affermare che un algoritmo **termina** significa assicurare che per ogni input la computazione finisca. Questi concetti coincidono in qualche caso, ma non in tutti. Ad esempio, per un algoritmo che enumera tutti i sottoinsiemi di un insieme A con una certa proprietà, diremo che è corretto se ogni $B \subseteq A$ che viene restituito possiede la proprietà cercata, che è completo se tutti i sottoinsiemi che possiedono la proprietà cercata vengono effettivamente elencati, e che termina se, indipendentemente da A, la computazione finisce in un tempo finito.

Analizzare gli algoritmi

Molti problemi che vedremo sono di tipo **funzionale**: trova il minimo, il massimo, restituisci l'input ordinato,. . . . In questi casi, distinguere tra le caratteristiche viste prima non è sempre conveniente nè naturale. In generale, noi ci riferiremo semplicemente alla correttezza di un algoritmo per un problema funzionale, e diremo appunto che esso è **corretto** se restituisce tutte e solamente le risposte giuste, e termina sempre. Il nostro modo di procedere, dunque, sará quello di mostrare la correttezza degli algoritmo che proponiamo, e studiarne la complessitá.

Analizzare gli algoritmi: correttezza nel caso iterativo

Per comprendere come dimostriamo formalmente la correttezza degli algoritmi iterativi, partiamo da un algoritmo ben conosciuto per ordinare un array con numeri interi, cioè *InsertionSort*, introdotto probabilmente la prima volta dal fisico John Mauchly, nel 1946.

Analizzare gli algoritmi: correttezza nel caso iterativo

```
proc InsertionSort (A)
  for (j = 2 to A.length)
    key = A[j]
    i = j − 1
    while ((i > 0) and (A[i] > key))
      A[i + 1] = A[i]
      i = i − 1
    A[i + 1] = key
```

A:

5	1	4	7	2	6
1	2	3	4	5	6

A.length = 6

key = 1

i j

già ordinato

5 5 4 7 2 6

A:

1	5	4	7	2	6

key = 4

i j

1 5 5 7 2 6

A:

1	4	5	7	2	6

i j

Analizzare gli algoritmi: correttezza nel caso iterativo

Vogliamo mostrare la correttezza dell'algoritmo. Per la **terminazione**, osserviamo che il ciclo **for** termina quando $j > A.length$, e il ciclo **while** è sempre vincolato tra $j - 1$ e 0, e che per ogni j, la variabile i, che inizia da un valore positivo, si decrementa sempra. Dunque la terminazione è garantita sia al livello del ciclo più esterno che di quello più interno. Per la **correttezza** del processo, usiamo la tecnica dell'**invariante**, che consiste nello stabilire una proprietà del ciclo principale (o di un ciclo dell'algoritmo) che sia vera prima della prima esecuzione, durante ogni esecuzione, e dopo l'ultima esecuzione del ciclo, e che implichi la correttezza. Nel nostro caso una **invariante** del ciclo piú esterno è che: $A[1, \ldots, j - 1]$ è sempre ordinato in maniera non decrescente; si noti che quando $j = A.length + 1$, l'algoritmo termina, verificando che $A[1, \ldots, n]$ è ordinato in maniera non decrescente.

Caso Base

Per dimostrare che l'invariante funziona, seguiamo il seguente ragionamento. **Prima della prima esecuzione** del ciclo **for**, il valore j è 2; ma $A[1, \ldots, 1]$ contiene solo una posizione ed è quindi ovviamente ordinato. **Dopo ogni esecuzione** l'elemento j-esimo è inserito in maniera ordinata nell'array (ordinato per ipotesi) $A[1, \ldots, j-1]$, ottenendo che, dopo questo inserimento, $A[1, \ldots, j]$ è ordinato. All'inizio della successiva iterazione, j viene incrementato, quindi la proprietà è vera proprio per $A[1, \ldots, j-1]$. **Dopo l'ultima esecuzione**, $j = A.length + 1$, e la proprietà dice precisamente che $A[1, \ldots, A.length]$ è ordinato.

Analizzare gli algoritmi: complessitá nel caso iterativo

Passiamo adesso allo studio della **complessitá** di *InsertionSort*.

A.length

proc ***InsertionSort*** (A)
- **for** $(j = 2$ **to** $A.length)$ $\Rightarrow n \cdot c_1$
  - $key = A[j]$ $\Rightarrow (n-1) \cdot c_2$
  - $i = j - 1$ $\Rightarrow (n-1) \cdot c_3$
  - **while** $((i > 0)$ **and** $(A[i] > key))$ $\Rightarrow c_4 \cdot \Sigma_{j=2}^{n} t_j$
    - $A[i+1] = A[i]$ $\Rightarrow c_5 \cdot \Sigma_{j=2}^{n} (t_j - 1)$
    - $i = i - 1$ $\Rightarrow c_6 \cdot \Sigma_{j=2}^{n} (t_j - 1)$
  - $A[i+1] = key$ $\Rightarrow (n-1) \cdot c_7$

$c_1, c_2, \ldots$ sono costanti; n è la dimensione dell'input; t_j va da 2 ad n e dipende da istanza a istanza. **Nel caso migliore** $t_j = 1$ per ogni $j = 2, \ldots, n$ e questo caso corrisponde all'input **gía ordinato** in partenza. **Nel caso peggiore** $t_j = j$ per ogni $j = 2, \ldots, n$ e corrisponde all'input **ordinato in ordine inverso** in partenza.

Analizzare gli algoritmi: complessitá nel caso iterativo

Nel caso migliore quindi:

$$T(n) = \underbrace{c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_7 \cdot (n-1)}_{C} + c_4 \cdot (n-1).$$

Quindi $\boxed{T(n)} = a \cdot n + b$ è una funzione *lineare* per qualche costante a, b.

— L — 10

CALCOLO NEL CASO MIGLIORE $(t_i = 1)$

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

$$c_1 n + c_2 n - c_2 + c_3 n - c_3 + c_7 n - c_7 + c_4 n - c_4 =$$

$$n(\underbrace{c_1 + c_2 + c_3 + c_7 + c_4}_{a}) + (\underbrace{-c_2 - c_3 - c_7 - c_4}_{b})$$

$$an + b$$

Analizzare gli algoritmi: complessitá nel caso iterativo

Nel caso peggiore invece, succede quanto segue. L'istruzione di controllo del **while** si esegue, ad ogni istanza del ciclo piú esterno, esattamente j volte; le due istruzioni interne, esattamente $j-1$ volte. Succede che:

$$\Sigma_{j=2}^{n} j = \frac{n \cdot (n+1)}{2} - 1$$

e che

$$\Sigma_{j=2}^{n} (j-1) = \frac{n \cdot (n-1)}{2},$$

e pertanto

$$T(n) = C + c_4 \cdot (\frac{n \cdot (n+1)}{2} - 1) + c_5 \cdot (\frac{n \cdot (n-1)}{2}) + c_6 \cdot (\frac{n \cdot (n-1)}{2}),$$

dove la parte C è rimasta come nel caso migliore.

Il risultato è quindi una funzione *quadratica* $\boxed{T(n)} = a \cdot n^2 + b \cdot n + c$ per qualche a, b, c.

CALCOLO NEL CASO PEGGIORE $(t_j = j)$ (1)

$$\sum_{j=2}^{n} j = \sum_{j=1}^{n} j - 1 = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n} (j-1) = \sum_{j=2}^{n} j - \sum_{j=2}^{n} 1 = \frac{n(n+1)}{2} - 1 - (n-1)$$

$$= \frac{n(n+1)}{2} - \frac{2}{2} - \frac{2(n-1)}{2}$$

$$= \frac{n(n+1) - 2 - 2n + 2}{2}$$

$$= \frac{n^2 + n - 2 - 2n + 2}{2}$$

$$= \frac{n^2 - n}{2} = \boxed{\frac{n(n-1)}{2}}$$

(2)

$$= c_1 n + c_2(n-1) + c_3(n-1) + c_7(n-1) +$$

$$c_4\left[\frac{n(n+1)}{2} - 1\right] + c_5\left[\frac{n(n-1)}{2}\right] + c_6\left[\frac{n(n-1)}{2}\right]$$

$$= c_1 n + c_2 n - c_2 + c_3 n - c_3 + c_7 n - c_7 +$$

$$c_4\left(\frac{n^2+n-2}{2}\right) + c_5\left(\frac{n^2-n}{2}\right) + c_6\left(\frac{n^2-n}{2}\right)$$

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

$$= \_\_\_\_\_ +$$

$$\frac{c_4 n^2}{2} + \frac{c_4 n}{2} - c_4 + \frac{c_5 n^2}{2} - \frac{c_5 n}{2} + \frac{c_6 n^2}{2} - \frac{c_6 n}{2}$$

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

Analizzare gli algoritmi: complessitá nel caso iterativo

Il nostro scopo era calcolare $T(n)$ cioè il numero di operazioni semplici effettuate per l'entrata di dimensione n, e lo abbiamo fatto **a meno di qualche costante**. Il caso migliore non è mai una buona scelta. Lo faremo sempre nel **caso peggiore** o nel **caso medio**, il quale a sua volta coincide nella maggior parte dei casi con il peggiore. In casi particolari faremo una analisi probabilistica nella speranza che il caso medio presenti un comportamento migliore.

Analizzare gli algoritmi: correttezza nel caso ricorsivo

Le tecniche che abbiamo visto per *InsertionSort* sono sostanzialmente utilizzabili per tutti gli algoritmi che non sono ricorsivi. In alcuni casi i conti possono diventare più complessi, ma non ci sono idee diverse o concetti più profondi che intervengono. Quando gli algoritmi sono particolarmente complessi, molto probabilmente sono basati su proprietá, lemmi, e teoremi che valgono per gli oggetti di cui si parla, e che, eventualmente, vanno dimostrati a parte. Quando dobbiamo analizzare la complessità di un algoritmo ricorsivo, invece, la questione cambia aspetto.

Per introdurre questi concetti, utilizziamo il ben noto algoritmo di ricerca binaria, *RecursiveBinarySearch*, la cui introduzione è associata a vari nomi, tra cui Thomas Hibbard, negli anni 60.

Analizzare gli algoritmi: correttezza nel caso ricorsivo

La chiamata iniziale è

RBS (A, 1, A.length, k)

proc ***RecursiveBinarySearch*** $(A, low, high, k)$
if $(low > high)$
  then return nil → caso base (negativo)
$mid = (high + low)/2$
if $(A[mid] = k)$ → ~~Primo~~ caso base (positivo)
  then return mid
if $(A[mid] < k)$
  then return $RecursiveBinarySearch(A, mid + 1, high, k)$
if $(A[mid] > k)$
  then return $RecursiveBinarySearch(A, low, mid - 1, k)$

Esempio:

$k = 20$

A:	13	6	7	10	12	14	20	23
	1	2	3	4	5	6	7	8

low (1), high (8)

mid (4)

low (4), low (5), high (8)

mid (6)

low (7), high (8)

mid (7) $\longrightarrow$ OUTPUT

Analizzare gli algoritmi: correttezza nel caso ricorsivo

L'algoritmo *RecursiveBinarySearch* serve a risolvere il seguente problema: dato un array, ordinato, ed una chiave k, restituire, se esiste, l'indice di k nell'array, altrimenti restituire **nil**. Questa parola chiave, **nil**, indica una locazione vuota di memoria; si tratta di una astrazione che useremo in maniera libera, anche quando parleremo di puntatori.
RecursiveBinarySearch è un algoritmo interessante perchè molto piú efficiente dell'idea naïve di scorrere l'array in cerca di k. L'idea è la stessa che abbiamo usato per risolvere in maniera efficiente il problema della moneta falsa. Come ne dimostriamo la correttezza?

Analizzare gli algoritmi: correttezza nel caso ricorsivo

Come nel caso iterativo, vogliamo assicurare la **terminazione** dell'algoritmo: ad ogni chiamata ricorsiva, la differenza tra *high* e *low* diminuisce. La condizione per effettuare un'altra chiamata ricorsiva è che $low < high$, cosa che deve necessariamente essere falsa, prima o poi, il che implica che ad un certo punto non ci saranno piú chiamate. Per la **correttezza**, utilizziamo la tecnica dell' **invariante induttiva**, che è una generalizzazione di quella dell'invariante. Invece di concentrarci su un ciclo, ci concentriamo su ció che accade **dopo ogni chiamata ricorsiva**. Nel caso di *RecursiveBinarySearch*, ad esempio, possiamo dire che: dopo ogni chiamata ricorsiva, se k è in A, allora si trova in $A[low, \ldots, high]$.

Analizzare gli algoritmi: correttezza nel caso ricorsivo

Le invarianti ricorsive si mostrano vere attraverso l'induzione, che, se vogliamo, è la faccia matematica della ricorsione.

- Nel **caso base**, cioè prima della prima chiamata ricorsiva, $low = 1$ e $high = n$; quindi se k è in A, è chiaramente in $A[low, \ldots, high]$.
- Nel **caso induttivo** si assume che k, se esiste, sia in $A[low, \ldots, high]$ (prima della prossima chiamata ricorsiva); poichè A è ordinato, e mid è l'indice mediano, se k esiste è $A[mid]$, oppure è alla sua sinistra (se $A[mid] > k$), oppure è alla sua destra (se $A[mid] < k$), e la successiva chiamata ricorsiva, se si effettua, cambia *low* o *high* in maniera da rispettare proprio questo principio. Dunque k, se esiste, si trova di nuovo in $A[low, \ldots, high]$ all'inizio della seguente chiamata ricorsiva, dove *low* o *high* è stato opportunamente cambiato.

Analizzare gli algoritmi: complessitá nel caso ricorsivo

Complessitá. Operare come in *InsertionSort* per studiare la complessitá di *RecursiveBinarySearch* non porta a nulla: tutte le operazioni semplici hanno complessità costante e non ci sono cicli. Ci sono, però, due chiamate ricorsive, che rendono difficile formalizzare il costo totale. $T(n)$, cioè la nostra funzione di complessità, è una **incognita**, della quale sappiamo le seguenti cose: costa una costante c **prima di richiamarsi**, che possiamo approssimare con 1, e nel **caso peggiore** si richiama esattamente una volta; poichè la variabile *mid* assume il valore dell'indice centrale dell'array arrotondato, eventualmente, all'intero inferiore, quando la procedura si richiama lo fa, al massimo, sulla **metà** degli elementi originali. Formalizzando:

$$T(n) = T(\frac{n}{2}) + 1$$

NEL CASO PEGGIORE

Diciamo che l'espressione che descrive $T(n)$ è una **ricorrenza**, e che questa si risolve trovando l'aspetto esplicito di $T(n)$.

Algoritmi e strutture dati

Notazione asintotica: principi

GIUSTA

Informalmente, data una funzione $T(n)$, diremo che $T(n) = \Theta(f(n))$ se e solo se si ottiene da $f(n)$ **eliminando tutti i termini di ordine inferiore al maggiore e tutte le sue costanti.** Quindi, per esempio, $T(n)$ di *InsertionSort* nel caso migliore è $\Theta(n)$, e, in maniera simile, $T(n)$ di *InsertionSort* nel caso peggiore è $\Theta(n^2)$. L'argomento che si usa per giustificare questa semplificazione è che, seppure è vero che $a \cdot n^2 + b \cdot n + c$ supera n^2, per istanze dove n cresce *molto*, le due funzioni diventano indistinguibili. Pertanto, è piú facile confrontare differenti algoritmi secondo il **loro comportamento asintotico nel caso peggiore**.

esempio:

$$T(n) = \cancel{a}n^2 + \cancel{bn} + \cancel{c} = \Theta(n^2)$$

$$T(n) = \cancel{a}n + \cancel{b} = \Theta(n)$$

Studiare la complessità

Facciamo un'ultima considerazione. Quali sono le operazioni che davvero contano quando si va a calcolare la complessitá? Osserviamo che la logica di un algoritmo è data unicamente dalle operazioni, appunto, **logiche**. Quindi, in realtá è naturale contare queste, invece che tutte le operazioni, perchè queste ultime danno un apporto costante, o comunque limitato dalle operazioni logiche sotto le quali sono inserite. A seconda del contesto, dunque, ci chiederemo quanti sono i **confronti**, o piú in generale quante sono le operazioni **logiche** di un certo algoritmo, piuttosto che semplicemente le operazioni. In particolare, quando calcoliamo la complessitá asintotica contiamo tutte le operazioni, e quando facciamo una analisi piú specifica ci concentriamo sulle operazioni logiche.

Notazione asintotica

A questo punto possiamo studiare con maggior dettaglio il concetto di **ordine di grandezza** di una funzione sui numeri naturali, introdotto come concetto intuitivo quando abbiamo visto la notazione $\Theta()$.

Nel caso di *InsertionSort* abbiamo detto che:

$$T(n) = a \cdot n^2 + b \cdot n + c = \Theta(n^2).$$

Ma cos'è $\Theta(f(n))$? Il nostro obbiettivo adesso è definire formalmente questa nozione.

Per definire $\Theta(f(n))$ prima di

$O(f(n))$

$\Omega(f(n))$

Notazione asintotica

Per una funzione $f(n)$ diremo, in primo luogo, che $f(n)$ è **limitata da** (o è un **"o" grande** di[1]) $g(n)$ (denotato $f(n) = O(g(n))$) se e solo se **esiste una costante** $c > 0$ tale che, per qualche n_0, **per tutti gli** $n \geq n_0$ è il caso che:

$$0 \leq f(n) \leq c \cdot g(n).$$

In maniera simile, per una funzione $f(n)$ diremo che $f(n)$ è **limitata dal basso da** (o è un **omega grande** di) $g(n)$ (denotato $f(n) = \Omega(g(n))$) se e solo se **esiste una costante** $c > 0$ tale che, per qualche n_0, **per tutti gli** $n \geq n_0$ è il caso che:

$$0 \leq c \cdot g(n) \leq f(n).$$

[1] Nelle ricerche su internet, ricordate che gli inglesi lo chiamano "Big Oh"

$$f(n) = O\big(g(n)\big) \iff \exists\, n_0, c \text{ t.c. } \forall n \geq n_0 \; f(n) \leq c\, g(n)$$

$$f(n) = \Omega\big(g(n)\big) \iff \exists\, n_0, c \text{ t.c. } \forall n \geq n_0 \; f(n) \geq c\, g(n)$$

ABBASTANZA GRANDE

$$\Downarrow$$

$$f(n) = \Theta\big(g(n)\big) \iff \exists\, n_0, c_1, c_2 \text{ t.c. } \forall n \geq n_0$$

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

SAREBBE PIÙ CORRETTO USARE $f(n) \in \begin{matrix} O(g(n)) \\ \Omega(g(n)) \\ \Theta(g(n)) \end{matrix}$

Notazione asintotica

$an^k = O(n^k)$

Nel caso piú semplice, abbiamo, ad esempio, che $f(n) = a \cdot n^k$ è tale che $f(n) = O(n^k)$, perchè, per ogni $n \geq 0$, si ha che esiste una costante c (che è proprio a) tale che $f(n) \leq c \cdot n^k$. In questo semplicissimo esempio, abbiamo anche che $f(n) = \Omega(n^k)$, per la stessa ragione.

Possiamo generalizzare questa proprietà?

Notazione asintotica

Il caso piú comune è quello dei polinomi algebrici di un certo grado k. Poichè noi consideriamo funzioni di complessità che emergono da algoritmi reali, $n > 0$ (la cardinalità dell'input è sempre positiva), cosí come tutte le costanti coinvolte. Quindi se ci limitiamo ai polinomi, in generale, avremo oggetti del tipo:

$$a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \ldots + a_1 \cdot n + a_0 = \Sigma_{i=0}^{k} a_i \cdot n^i$$

Per questi oggetti è immediato dimostrare, come è anche intuitivo, che:

$$\Sigma_{i=0}^{k} a_i \cdot n^i = O(n^k).$$

Infatti, sotto l'ipotesi che tutte le costanti sono positive, otteniamo:

$$\begin{array}{lll} \Sigma_{i=0}^{k} a_i \cdot n^i & \leq \Sigma_{i=0}^{k} a_i \cdot n^k & \quad \textit{maggiorazione} \\ & = (\Sigma_{i=0}^{k} a_i) \cdot n^k & \\ & = c \cdot n^k & \end{array}$$

Dimostro che

$$a_k n^k + a_{k-1} n^{k-1} + a_{k-2} n^{k-2} + \dots + a_0 = O(n^k)$$

$$\sum_{i=0}^{k} a_i n^i \le \sum_{i=0}^{k} a_i n^k = n^k \underbrace{\sum_{i=0}^{k} a_i}_{\text{costante}}$$

$$\le c n^k = O(n^k)$$

In maniera simile, possiamo dire che:

$$\Sigma_{i=0}^{k} a_i \cdot n^i = \Omega(n^k).$$

Infatti:

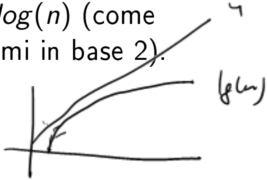
$$\begin{aligned}\Sigma_{i=0}^{k} a_i \cdot n^i &\geq a_k \cdot n^k \quad \textit{minorazione} \\ &= c \cdot n^k\end{aligned}$$

E ambedue queste disequazioni valgono per ogni $n > 0$, quindi basta prendere $n_0 = 0$ per ottenere la definizione esatta in entrambi i casi. Queste considerazioni valgono anche quando le costanti non sono tutte positive; la dimostrazione è leggermente piú complessa.

$$\Rightarrow \sum_{i=0}^{k} a_i n^i = \Theta(n^k)$$

Notazione asintotica

Naturalmente, $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$ **non** sono definizioni equivalenti. Prendiamo il caso, ad esempio, di $f(n) = log(n)$ (come sempre, dove non specificato altrimenti, usiamo logaritmi in base 2). Succede, da un lato, che:



$$log(n) = O(n).$$

Questo lo vediamo per induzione su n. Se $n = 1$, allora $log(n) = 0 \leq 1 \cdot 1$, cioè la definizione vale per $c = 1$. Assumendo quindi che $log(n) \leq n$, dimostriamo che $log(n+1) \leq n+1$:

$$\begin{aligned} log(n+1) &\leq log(2 \cdot n) && \textit{maggiorazione, log è monotona} \\ &= log(2) + log(n) && \textit{proprietá dei logaritmi} \\ &\leq n+1 && \textit{ipotesi induttiva} \end{aligned}$$

Ma chiaramente non è vero che $log(n) \geq c \cdot n$ per $n \geq n_0$ per nessuna scelta di c e di n_0. Quindi $log(n) \neq \Omega(n)$.

Notazione asintotica

$f(n) = \Theta(g(n)) \iff f(n) = O(g(n))$

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

Arriviamo adesso alla nozione che avevamo al principio. Per una funzione $f(n)$, diremo che $f(n)$ è **dello stesso ordine di** $g(n)$ se e solo se **esistono due costanti** $c_1, c_2 > 0$ tali che, per qualche n_0, **per tutti gli** $n \geq n_0$ è il caso che:

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n).$$

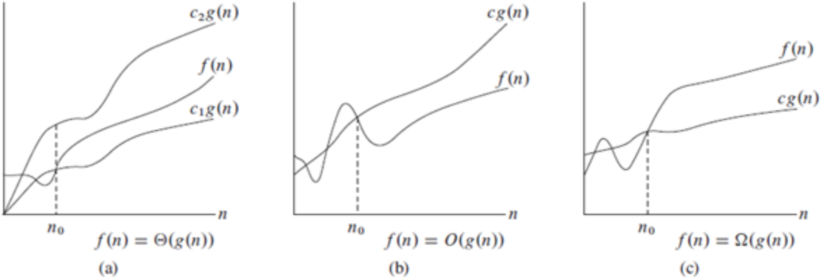
In altre parole l'insieme $\Theta(g(n))$ contiene tutte e sole quelle funzioni $f(n)$ tali che, a partire da un certo momento in poi (per tutti gli $n \geq n_0$), il valore di $f(n)$ è descritto dal valore $g(n)$ a meno di una costante. Quindi, asintoticamente, $f(n)$ si comporta come $g(n)$.

Chiaramente:

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

se e solo se:

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



Notazione asintotica

Per confrontare gli ordini di grandezza di funzioni diverse, usiamo la notazione $o()$ e $\omega()$ (**o piccolo** e **omega piccolo**).

Diremo quindi che $f(n) = o(g(n))$ (**è di un ordine infieriore a**) se e solo se per ogni costante $c > 0$, esiste una costante n_0 tale che, per ogni $n > n_0$ si verifica che $0 \leq f(n) \leq c \cdot g(n)$. Questo si puó vedere in maniera piú semplice: $lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Analogamente, $f(n) = \omega(g(n))$ (**è di un ordine superiore a**) se e solo se per ogni costante $c > 0$, esiste una costante n_0 tale che, per ogni $n > n_0$ si verifica che $0 \leq c \cdot g(n) \leq f(n)$. Piú semplicemente: $lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

ES: $n = o(n^2)$ poiché $\lim_{n \to \infty} \frac{n}{n^2} = \frac{1}{n} \to 0$

Conclusione

Il livello di dettaglio utilizzato in questi esempi è nettamente superiore a quello che utilizziamo normalmente. *InsertionSort* e *RecursiveBinarySearch* sono algoritmi particolarmente semplici che abbiamo utilizzato allo scopo di introdurre queste tecniche; nel futuro, analizzeremo algoritmi anche molto piú complessi, ad un livello di dettaglio inferiore. Inoltre, la complessità di un problema è strettamente legata alla complessità di un algoritmo che lo risolve, ma dobbiamo fare le dovute precisazioni. Dato un algoritmo che risolve un problema con complessità, nel caso peggiore, $O(f(n))$ oppure $\Theta(f(n))$, possiamo dire che il problema stesso ha complessità $O(f(n))$; in assenza di una dimostrazione che quello è il miglior metodo possibile di soluzione, non possiamo però dire qualcosa di più stretto sul problema. Allo stesso modo, la complessità nel caso medio di un algoritmo che risolve un problema non fornisce informazioni sulla complessità del problema che risolve, se non il fatto che il problema stesso è in effetti risolvibile.

OSSERVAZIONI

1) Si confrontano solo algoritmi x lo stesso problema nello stesso caso (peggiore, medio, migliore)

2) La complessità di un problema è limitata dall'alto da ognuno degli algoritmi che lo risolvono valutato nel caso peggiore

PROPRIETÀ DI O, Ω, Θ CHE DIP. DALLE DEFINIZIONI

$$\text{Sia } * \in \{O, \Omega, \Theta\}$$

① $*\big(f(n)+g(n)\big) = *\big(\max\{f(n), g(n)\}\big)$

② $*\big(f(n)+g(n)\big) = *\big(f(n)\big)$ quando $g(n) = o\big(f(n)\big)$

③ $*\big(f(n)+g(n)\big) = *\big(f(n)\big) + *\big(g(n)\big)$ quando $f(n), g(n)$ sono positive

ESERCIZIO 2:

Sotto queste ipotesi, $t_j = k$ (vedi slide 9), dunque

$$T(n) = nc_1 + (n-1)c_2 + (n-1)c_3 + (k+1)c_4 + kc_5 + kc_6 + (n-1)c_7$$

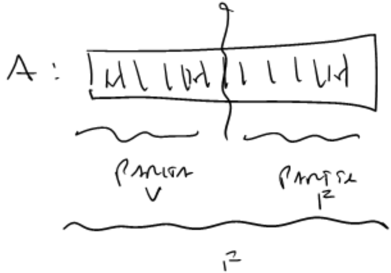
$$= nc_1 + nc_2 - c_2 + nc_3 - c_3 + kc_4 + c_4 + kc_5 + kc_6 + nc_7 - c_7$$

$$= \underbrace{n(c_1 + c_2 + c_3 + c_7)}_{a} + \underbrace{(-c_2 - c_3 + kc_4 + kc_5 + kc_6 - c_7)}_{b}$$

$$= an + b = \Theta(n)$$

$$\boxed{A}$$

ESERCIZIO 5



Invariante: dopo ogni chiamata ricorsiva, si restituiscono due parti del ~~X~~ di accumulo

10

$\boxed{\text{Esercizio 6}}$

$$T(n) = 2T\left(\frac{n}{2}\right) + \begin{array}{l} 1 \\ c \\ \Theta(1) \\ \Theta(c) \end{array}$$

$\boxed{1B}$

ESERCIZIO 12

VERIFICO CHE $\boxed{G}$ E' CORRETTA

$$\begin{cases} 3x^2+5x+1 \geq \frac{3}{4}x^2 \\ 3x^2+5x+1 \leq \frac{21}{4}x^2 \end{cases}$$

DEVE SUCCEDERE CHE LE SOLUZIONI ~~SIANO~~ COMPRENDANO L'INTERVALLO $[5, \infty[$

$\to$ $12x^2+20x+4 \geq 3x^2$

$9x^2+20x+4 \geq 0$

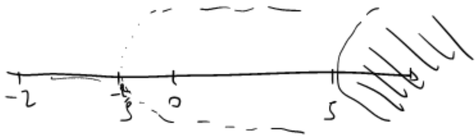
EQ. CORR. $x_{12} = \frac{-20 \pm \sqrt{400-16\cdot 9}}{18}$

$-\frac{36}{18} = -2$

$-\frac{4}{18} = -\frac{2}{9}$

$\downarrow$

SOL. SISTEMA



Esercizio 17

Si chiede di studiare $\lg(n!)$

$$n! = 1 \cdot 2 \cdot 3 \cdots n$$

$$\lg(n!) = \lg(1 \cdot 2 \cdot 3 \cdots n) = \lg(1) + \lg(2) \ldots + \lg(n)$$

① $\lg(n!) \leq \lg(n) + \lg(n) + \ldots + \lg(n) \leq n\lg(n) = O(n\lg(n))$

② $\lg(n!) \geq \underbrace{\lg\left(\frac{n}{2}\right) + \lg\left(\frac{n}{2}\right) + \ldots \lg\left(\frac{n}{2}\right)}_{n/2} \geq \frac{n}{2} \lg\left(\frac{n}{2}\right) = \frac{n}{2}(\lg(n) - 2) = \frac{1}{2} n\lg(n) - n = \Omega(n\lg(n))$

$\boxed{C}$

$= \Theta(n\lg(n))$

$\lim_{n\to\infty} \frac{n}{n\lg(n)} = \lim_{n\to\infty} \frac{1}{\lg(n)} \to 0$

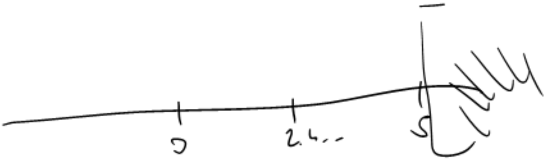
... CONTINUA

$12u^2+20u+4 \le 21u^2$

$9u^2-20u-4 \ge 0$ EQ. CONNESSA $u_{1,2} = \dfrac{20 \pm \sqrt{400 + 9 \cdot 16}}{18}$

$\dfrac{20+23...}{18} = 2.4=$

$\dfrac{20-23...}{18}$??



DIV