



2) Complessità asintotica

Per poter valutare l'efficienza di un algoritmo, così da poterlo confrontare con algoritmi diversi che risolvono lo stesso problema, bisogna essere in grado di valutare l'ordine di grandezza del suo tempo di esecuzione e delle sue necessità in termini di memoria. In particolare, tale valutazione ha senso quando la dimensione dell'input è sufficientemente grande.

La disciplina che si occupa della quantificazione di tali grandezze è la **complessità computazionale asintotica**:

- complessità computazionale: indica che ci occupiamo di valutare il "costo di esecuzione" dell'algoritmo;
- asintotica: indica che siamo interessati a risultati di tipo asintotico, ossia validi per grandi dimensioni dell'input.

Salvo ove sia diversamente specificato, la grandezza che viene valutata è il tempo di esecuzione dell'algoritmo.

Prima di entrare nel vivo della valutazione della complessità computazionale asintotica degli algoritmi è necessario introdurre alcune definizioni.

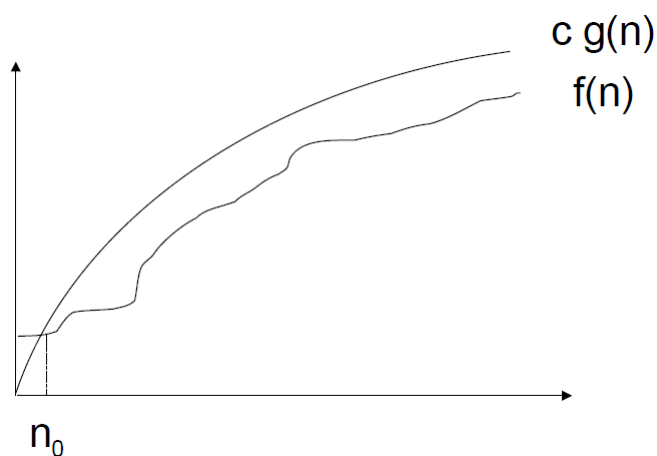
2.1 Notazione O (limite asintotico superiore)

Date due funzioni $f(n)$, $g(n) \geq 0$ si dice che

$$f(n) \text{ è un } O(g(n))$$

se esistono due costanti c ed n_0 tali che

$$0 \leq f(n) \leq c g(n) \text{ per ogni } n \geq n_0$$





Esempio 2.1

Sia $f(n) = 3n + 3$

$f(n)$ è un $O(n^2)$ in quanto, posto $c = 6$, $cn^2 \geq 3n + 3$ per ogni $n \geq 1$.

Ma $f(n)$ è anche un $O(n)$ in quanto $cn \geq 3n + 3$ per ogni $n \geq 1$ se $c \geq 6$, oppure per ogni $n \geq 3$ se $c \geq 4$.

E' facile convincersi che, data una funzione $f(n)$, esistono infinite funzioni $g(n)$ per cui $f(n)$ risulta un $O(g(n))$. Come sarà più chiaro in seguito, ci interessa tentare di determinare la funzione $g(n)$ che meglio approssima la funzione $f(n)$ dall'alto o, informalmente, la più piccola funzione $g(n)$ tale che $f(n)$ sia $O(g(n))$.

Esempio 2.2

Sia $f(n) = n^2 + 4n$

Ebbene, $f(n)$ è un $O(n^2)$ in quanto $cn^2 \geq n^2 + 4n$ per ogni n se $c \geq 5$, oppure per ogni $n \geq 4/(c-1)$ se $c > 1$.

Esempio 2.3

Sia $f(n)$ un polinomio di grado m : $f(n) = \sum_{i=0}^m a_i n^i$, con $a_m > 0$

Dimostriamo, per induzione su m , che $f(n)$ è un $O(n^m)$.

Casi base

- $m = 0$: $f(n) = a_0$, quindi $f(n)$ è un $O(1)$ per qualunque n , per ogni $c \geq a_0$
- $m = 1$: $f(n) = a_0 + a_1 n$, quindi $f(n)$ è un $O(n)$ per qualunque n , per ogni $c \geq a_0 + a_1$

Ipotesi induttiva

$h(n) = \sum_{i=0}^{m-1} a_i n^i$ è un $O(n^{m-1})$ se $c \geq \sum_{i=0}^{m-1} a_i$

Passo induttivo

Dobbiamo dimostrare che

$$f(n) = \sum_{i=0}^m a_i n^i \text{ è un } O(n^m)$$

cioè che

$$c'n^m \geq \sum_{i=0}^m a_i n^i \text{ se } c' \geq \sum_{i=0}^m a_i$$

Per l'ipotesi induttiva:

- $f(n)$ si può scrivere come $f(n) = h(n) + a_m n^m$
- $cn^{m-1} \geq h(n)$ per ogni $c \geq \sum_{i=0}^{m-1} a_i$



Ora:

$$f(n) = h(n) + a_m n^m \leq cn^{m-1} + a_m n^m \leq cn^m + a_m n^m = (c + a_m)n^m$$

Ponendo $c' = c + a_m$ si ha la tesi.

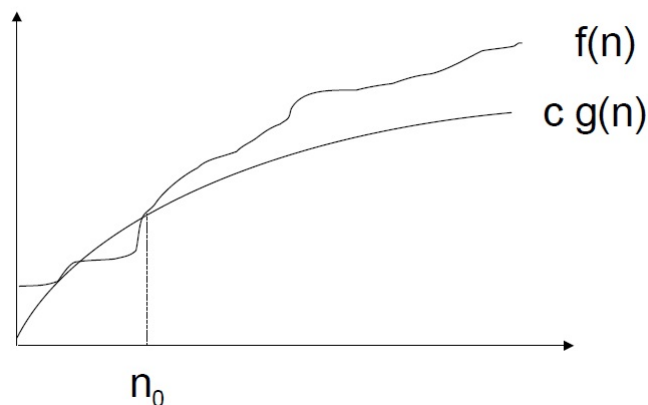
2.2 Notazione Ω (limite asintotico inferiore)

Date due funzioni $f(n), g(n) \geq 0$ si dice che

$$f(n) \text{ è un } \Omega(g(n))$$

se esistono due costanti c ed n_0 tali che

$$f(n) \geq c g(n) \text{ per ogni } n \geq n_0$$



Esempio 2.4

Sia $f(n) = 2n^2 + 3$

$f(n)$ è un $\Omega(n)$ in quanto, posto $c = 1$, $2n^2 + 3 \geq cn$ per qualunque n

Ma $f(n)$ è anche un $\Omega(n^2)$ in quanto $2n^2 + 3 \geq cn^2$ per ogni n , se $c \leq 2$.

Esempio 2.5

Sia $f(n)$ un polinomio di grado m : $f(n) = \sum_{i=0}^m a_i n^i$, con $a_m > 0$

La dimostrazione che $f(n)$ è un $\Omega(n^m)$ è analoga alla dimostrazione che $f(n)$ è un $O(n^m)$ e perciò viene lasciata come esercizio.



Abbiamo visto come, in entrambe le notazioni espresse in precedenza, per ogni funzione $f(n)$ sia possibile trovare più funzioni $g(n)$. In effetti $O(g(n))$ e $\Omega(g(n))$ sono **insiemi di funzioni**, e dire “ $f(n)$ è un $O(g(n))$ ” oppure “ $f(n) = O(g(n))$ ” ha il significato di “ $f(n)$ appartiene a $O(g(n))$ ”.

Tuttavia, poiché i limiti asintotici ci servono per stimare con la maggior precisione possibile la complessità di un algoritmo, vorremmo trovare – fra tutte le possibili funzioni $g(n)$ – quella che più si avvicina a $f(n)$.

Per questo cerchiamo la più piccola funzione $g(n)$ per determinare O e la più grande funzione $g(n)$ per determinare Ω .

La definizione che segue formalizza questo concetto intuitivo.

2.3 Notazione Θ (limite asintotico stretto)

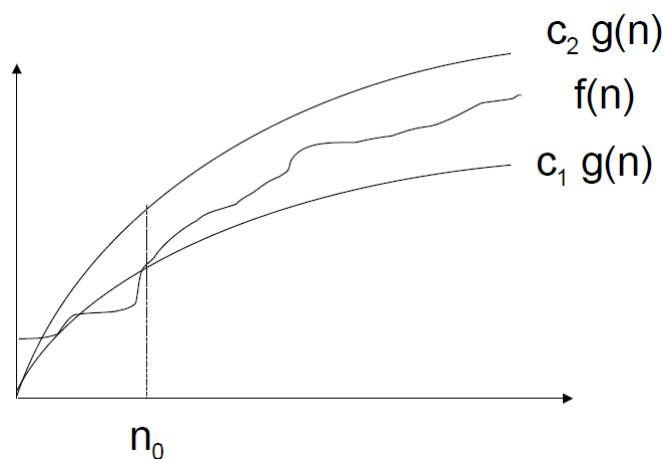
Date due funzioni $f(n)$, $g(n) \geq 0$ si dice che

$$f(n) \text{ è un } \Theta(g(n))$$

se esistono tre costanti c_1 , c_2 ed n_0 tali che

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

In altre parole, $f(n)$ è $\Theta(g(n))$ se è contemporaneamente $O(g(n))$ e $\Omega(g(n))$.



Esempio 2.6

Sia $f(n) = 3n + 3$

$f(n)$ è un $\Theta(n)$ ponendo, ad esempio, $c_1 = 3$, $c_2 = 4$, $n_0 = 3$.



Esempio 2.7

Sia $f(n)$ un polinomio di grado m : $f(n) = \sum_{i=0}^m a_i n^i$, con $a_m > 0$

La dimostrazione che $f(n)$ è un $\Theta(n^m)$ discende dagli esempi 2.3 e 2.5.

2.4 Algebra della notazione asintotica

Per semplificare il calcolo della complessità asintotica degli algoritmi si possono sfruttare delle semplici regole che dapprima enunciamo chiarendole con degli esempi, ed in un secondo momento dimostriamo.

Regole sulle costanti moltiplicative

1A: Per ogni $k > 0$ e per ogni $f(n) \geq 0$, se $f(n)$ è un $O(g(n))$ allora anche $k f(n)$ è un $O(g(n))$.

1B: Per ogni $k > 0$ e per ogni $f(n) \geq 0$, se $f(n)$ è un $\Omega(g(n))$ allora anche $k f(n)$ è un $\Omega(g(n))$.

1C: Per ogni $k > 0$ e per ogni $f(n) \geq 0$, se $f(n)$ è un $\Theta(g(n))$ allora anche $k f(n)$ è un $\Theta(g(n))$.

Informalmente, queste tre regole si possono riformulare dicendo che le costanti moltiplicative si possono ignorare.

Regole sulla commutatività con la somma

2A: Per ogni $f(n)$, $d(n) > 0$, se $f(n)$ è un $O(g(n))$ e $d(n)$ è un $O(h(n))$ allora $f(n)+d(n)$ è un $O(g(n)+h(n)) = O(\max(g(n), h(n)))$.

2B: Per ogni $f(n)$, $d(n) > 0$, se $f(n)$ è un $\Omega(g(n))$ e $d(n)$ è un $\Omega(h(n))$ allora $f(n)+d(n)$ è un $\Omega(g(n)+h(n)) = \Omega(\max(g(n), h(n)))$.

2C: Per ogni $f(n)$, $d(n) > 0$, se $f(n)$ è un $\Theta(g(n))$ e $d(n)$ è un $\Theta(h(n))$ allora $f(n)+d(n)$ è un $\Theta(g(n)+h(n)) = \Theta(\max(g(n), h(n)))$.

Informalmente, queste tre regole si possono riformulare dicendo che le notazioni asintotiche commutano con l'operazione di somma.

Regole sulla commutatività col prodotto

3A: Per ogni $f(n)$, $d(n) > 0$, se $f(n)$ è un $O(g(n))$ e $d(n)$ è un $O(h(n))$ allora $f(n)d(n)$ è un $O(g(n)h(n))$.

3B: Per ogni $f(n)$, $d(n) > 0$, se $f(n)$ è un $\Omega(g(n))$ e $d(n)$ è un $\Omega(h(n))$ allora $f(n)d(n)$ è un $\Omega(g(n)h(n))$.



3C: Per ogni $f(n)$, $d(n) > 0$, se $f(n)$ è un $\Theta(g(n))$ e $d(n)$ è un $\Theta(h(n))$ allora $f(n)d(n)$ è un $\Theta(g(n)h(n))$.

Informalmente, queste tre regole si possono riformulare dicendo che le notazioni asintotiche commutano con l'operazione di prodotto.

Prima di dimostrare le regole enunciate sopra, facciamo alcuni esempi.

Esempio 2.8

Trovare il limite asintotico stretto per $f(n) = 3n2^n + 4n^4$

$$3n2^n + 4n^4 = \Theta(n)\Theta(2^n) + \Theta(n^4) = \Theta(n2^n) + \Theta(n^4) = \Theta(n2^n).$$

Esempio 2.9

Trovare il limite asintotico stretto per $f(n) = 2^{n+1}$

$$2^{n+1} = 2 \cdot 2^n = \Theta(2^n).$$

Esempio 2.10

Trovare il limite asintotico stretto per $f(n) = 2^{2^n}$

$$2^{2^n} = \Theta(2^{2^n}).$$

Questo esempio ci permette di notare che le **costanti moltiplicative si possono ignorare solo se non sono all'esponente**.

Dimostriamo ora tutte le regole precedentemente enunciate.

Regola 1A. Per ogni $k > 0$ e per ogni $f(n) \geq 0$, se $f(n)$ è un $O(g(n))$ allora anche $k f(n)$ è un $O(g(n))$.

Dim. Per ipotesi, $f(n)$ è un $O(g(n))$ quindi esistono due costanti c ed n_0 tali che:

$$f(n) \leq cg(n) \text{ per ogni } n \geq n_0$$

Ne segue che:

$$kf(n) \leq kcg(n)$$

Questo prova che, prendendo kc come nuova costante c' e mantenendo lo stesso n_0 , $kf(n)$ è un $O(g(n))$.

CVD



Regola 2A. Per ogni $f(n)$, $d(n) > 0$, se $f(n)$ è un $O(g(n))$ e $d(n)$ è un $O(h(n))$ allora $f(n)+d(n)$ è un $O(g(n)+h(n)) = O(\max(g(n), h(n)))$.

Dim. Se $f(n)$ è un $O(g(n))$ e $d(n)$ è un $O(h(n))$ allora esistono quattro costanti c' e c'' , n'_0 ed n''_0 tali che:

$$f(n) \leq c'g(n) \text{ per ogni } n \geq n'_0 \text{ e } d(n) \leq c''h(n) \text{ per ogni } n \geq n''_0$$

allora:

$$f(n) + d(n) \leq c'g(n) + c''h(n) \leq \max(c', c'')(g(n) + h(n)) \text{ per ogni } n \geq \max(n'_0, n''_0)$$

Da ciò segue che $f(n) + d(n)$ è un $O(g(n)+h(n))$.

Infine:

$$\max(c', c'')(g(n) + h(n)) \leq 2 \max(c', c'') \max(g(n), h(n)).$$

Ne segue che $f(n) + d(n)$ è un $O(\max(g(n), h(n)))$.

CVD

Regola 3A. Per ogni $f(n)$, $d(n) > 0$, se $f(n)$ è un $O(g(n))$ e $d(n)$ è un $O(h(n))$ allora $f(n)d(n)$ è un $O(g(n)h(n))$.

Dim. Se $f(n)$ è un $O(g(n))$ e $d(n)$ è un $O(h(n))$ allora esistono quattro costanti c' e c'' , n'_0 ed n''_0 tali che:

$$f(n) \leq c'g(n) \text{ per ogni } n \geq n'_0 \text{ e } d(n) \leq c''h(n) \text{ per ogni } n \geq n''_0$$

allora:

$$f(n)d(n) \leq c'c''g(n)h(n) \text{ per ogni } n \geq \max(n'_0, n''_0)$$

Da ciò segue che $f(n)d(n)$ è un $O(g(n)h(n))$.

CVD

Le dimostrazioni delle altre regole che coinvolgono le notazioni Ω e Θ sono lasciate per esercizio.

2.5 Valutazione della complessità computazionale di un algoritmo

Vediamo ora come calcolare effettivamente la complessità computazionale di un algoritmo, adottando il criterio della misura di costo uniforme descritto nel par. 1.4.2.

Prima di procedere, facciamo due importanti considerazioni.

Innanzitutto, osserviamo che è ragionevole pensare che la complessità computazionale, intesa come funzione che rappresenta il tempo di esecuzione di un algoritmo, sia una funzione monotona non decrescente della dimensione dell'input. Questa osservazione ci conduce a constatare che, prima di passare al calcolo della complessità, bisogna definire quale sia la



dimensione dell'input. Trovare questo parametro è, di solito, abbastanza semplice: in un algoritmo di ordinamento esso sarà il numero di dati, in un algoritmo che lavora su una matrice sarà il numero di righe e di colonne, in un algoritmo che opera su alberi sarà il numero di nodi, ecc.. Tuttavia, vi sono casi in cui l'individuazione del parametro non è banale; in ogni caso, è necessario stabilire quale sia la variabile (o le variabili) di riferimento *prima* di accingersi a calcolare la complessità.

In secondo luogo, vogliamo sottolineare che la notazione asintotica viene sfruttata pesantemente per il calcolo della complessità computazionale degli algoritmi, quindi - in base alla definizione stessa - tale complessità computazionale potrà essere ritenuta valida *solo* asintoticamente.

In effetti, esistono degli algoritmi che per dimensioni dell'input relativamente piccole hanno un certo comportamento, mentre per dimensioni maggiori un altro. Sottolineiamo che siamo interessati all'andamento asintotico della complessità.

Per poter valutare la complessità di un algoritmo, esso deve essere formulato in un modo che sia chiaro, sintetico e non ambiguo.

Si adotta il cosiddetto **pseudocodice**, che è una sorta di linguaggio di programmazione "informale" nell'ambito del quale:

- si impiegano, come nei linguaggi di programmazione, i costrutti di controllo (for, if then else, while, ecc.);
- si impiega il linguaggio naturale per specificare le operazioni;
- si ignorano i problemi di ingegneria del software;
- si omette la gestione degli errori, al fine di esprimere solo l'essenza della soluzione.

Non esiste una notazione universalmente accettata per lo pseudocodice. In queste dispense (come del resto nel libro di testo) useremo l'indentazione per rappresentare i diversi livelli dei blocchi di istruzioni, useremo il simbolo \leftarrow per indicare un'assegnazione, il simbolo $=$ per verificare che il contenuto di 2 variabili sia lo stesso e il simbolo \neq per verificare che il contenuto di 2 variabili sia differente.

Le regole generali che si adottano per valutare la complessità computazionale di un algoritmo sono le seguenti:

- le istruzioni elementari (operazioni aritmetiche, lettura del valore di una variabile, assegnazione di un valore a una variabile, valutazione di una condizione logica su un numero costante di operandi, stampa del valore di una variabile, ecc.) hanno complessità $\Theta(1)$;



- l'istruzione `if (condizione) then istruzione1 else istruzione2` ha complessità pari al costo di verifica della condizione (di solito costante) più il massimo delle complessità di `istruzione1` e `istruzione2`;
- le istruzioni iterative (o iterazioni: cicli `for`, `while` e `repeat`) hanno una complessità pari alla somma delle complessità massime di ciascuna delle iterazioni (compreso il costo di verifica della condizione). Se tutte le iterazioni hanno la stessa complessità massima, allora la complessità dell'iterazione è pari al prodotto della complessità massima di una singola iterazione per il numero di iterazioni; in entrambi i casi è comunque necessario stimare il numero delle iterazioni. Si osservi che la condizione viene valutata una volta in più rispetto al numero delle iterazioni, poiché l'ultima valutazione, che dà esito negativo, è quella che fa terminare l'iterazione;
- la complessità dell'algoritmo è pari alla somma delle complessità delle istruzioni che lo compongono.

Si osservi che un dato algoritmo potrebbe avere tempi di esecuzione (e quindi complessità) diversi a seconda dell'input. In tal caso, per fare uno studio esauriente della sua complessità, bisogna valutare prima quali siano i cosiddetti casi migliore e peggiore, cioè cercare di capire quale input sia particolarmente vantaggioso, ai fini della complessità dell'algoritmo, e quale invece svantaggioso.

Per avere un'idea di quale sia il tempo di esecuzione atteso di un algoritmo, a prescindere dall'input, è chiaro che è necessario prendere in considerazione il caso peggiore, cioè la situazione che porta alla computazione più onerosa. Nel contempo, però, vorremmo essere il più precisi possibile e quindi, nel contesto del caso peggiore, cerchiamo di calcolare la complessità in termini di notazione asintotica Θ . Laddove questo non sia possibile, essa dovrà essere approssimata per difetto (tramite la notazione Ω) e per eccesso (tramite la notazione O).



Esempio 2.11

Calcolo del massimo in un vettore disordinato contenente n valori.

Funzione Trova_Max (A: vettore)

```
1   max ← A[1]                               Θ(1)
2   for i = 2 to n do                         (n - 1) iterazioni più Θ(1)
3       if A[i] > max                         Θ(1)
4           then max ← A[i]                   Θ(1)
5   stampa max                                Θ(1)
```

La complessità dell'istruzione 1 è $\Theta(1)$.

L'iterazione viene eseguita $(n - 1)$ volte, e ciascuna iterazione (costituita dalle istruzioni 2, 3 e 4) ha complessità $\Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$ poiché l'incremento del contatore (istr. 2), la valutazione della condizione (istr. 3) e l'assegnazione (istr. 4) sono istruzioni elementari.

La complessità dell'istruzione 5 è $\Theta(1)$.

Quindi, detta $T(n)$ la complessità temporale di questo algoritmo, essa vale:

$$T(n) = \Theta(1) + (n - 1) \Theta(1) + \Theta(1) = \Theta(n)$$

Concludiamo questo argomento mostrando come variano i tempi di esecuzione di un algoritmo in funzione della sua complessità temporale.

Ipotizziamo di disporre di un sistema di calcolo in grado di effettuare una operazione elementare in un nanosecondo (10^9 operazioni al secondo), e supponiamo che la dimensione del problema sia $n = 10^6$ (un milione):

- complessità $O(n)$ - tempo di esecuzione: 1 millesimo di secondo;
- complessità $O(n \log n)$ - tempo di esecuzione: 20 millesimi di secondo;
- complessità $O(n^2)$ - tempo di esecuzione: 1000 secondi = 16 minuti e 40 secondi.

C'è un'altra situazione interessante da considerare: che succede se la complessità cresce esponenzialmente, ad esempio quando è $O(2^n)$?

E' abbastanza ovvio che i tempi di esecuzione diventano rapidamente proibitivi: un tale tipo di problema su un input di dimensione $n = 100$ richiede per la sua soluzione mediante il sistema di calcolo di cui sopra ben $1,26 \cdot 10^{21}$ secondi, cioè circa $3 \cdot 10^{13}$ anni.



Si può pensare che l'avanzamento tecnologico, per quanto formidabile, possa rendere abbordabile un tale problema? Purtroppo la risposta è no. Infatti, poniamoci la seguente domanda: supponendo di avere un calcolatore estremamente potente che riesce a risolvere un problema di dimensione $n = 1000$, avente complessità $O(2^n)$, in un determinato tempo T , quale dimensione $n' = n + x$ del problema riusciremmo a risolvere nello stesso tempo utilizzando un calcolatore mille volte più veloce?

Possiamo scrivere la seguente uguaglianza:

$$T = \frac{2^{1000} \text{ operazioni}}{10^k \text{ operazioni al secondo}} = \frac{2^{1000+x} \text{ operazioni}}{10^{k+3} \text{ operazioni al secondo}}$$

Si ha quindi:

$$\frac{2^{1000+x}}{2^{1000}} = 2^x = \frac{10^{k+3}}{10^k} = 10^3 = 1000$$

Ossia

$$x = \log 1000 \approx 10$$

Dunque, con un calcolatore mille volte più veloce riusciremmo solo a risolvere, nello stesso tempo, un problema di dimensione 1010 anziché di dimensione 1000.

In effetti esiste un'importantissima branca della teoria della complessità che si occupa proprio di caratterizzare i cosiddetti problemi **intrattabili**, ossia quei problemi la cui complessità computazionale è tale per cui essi non sono né saranno mai risolvibili per dimensioni realistiche dell'input.