Dispense sulla Complessità Computazionale

Complessità Computazionale

Premessa

Un aspetto importante che non può essere trascurato nella progettazione di un algoritmo è la caratterizzazione dell'efficienza con la quale l'algoritmo stesso viene eseguito su un elaboratore. L'efficienza è, tra l'altro, da intendersi non solo come tempo di esecuzione, ma anche in funzione dell'utilizzo delle altre risorse del sistema di elaborazione, come, ad esempio, la memoria centrale.

In questo capitolo viene introdotto un modello per la misura dell'efficienza di un programma. Il modello permette di caratterizzare l'efficienza di esecuzione (*complessità computazionale* o *temporale*) e l'efficienza in termini di memoria impiegata (*complessità spaziale*) in maniera indipendente dal sistema di elaborazione sul quale si intende eseguire il programma in oggetto.

La misura dell'efficienza

L'efficienza di un algoritmo (e da ora in poi, se non esplicitamente indicato, parleremo indistintamente di efficienza spaziale o di efficienza computazionale) non può essere misurata semplicemente, in quanto dipendente da un numero elevatissimo di fattori. Dato un programma scritto in linguaggio sorgente l'efficienza dell'esecuzione dipende dal compilatore impiegato che effettua una traduzione in codice macchina che può essere più o meno efficiente, dalla struttura del sistema di elaborazione che può risultare più adeguata ad eseguire rapidamente alcune classi di istruzioni, dalla potenza della CPU del sistema di elaborazione, e via di seguito.

E' quindi importante notare esplicitamente che il tentativo di caratterizzare l'efficienza non può che prescindere dalla architettura di sistema di elaborazione adottato, e concentrarsi sugli aspetti che invece sono strettamente legati alla natura e struttura dell'algoritmo prescelto.

Nel seguito di questo paragrafo tenteremo di definire in che ottica avviene la caratterizzazione dell'efficienza e nei paragrafi seguenti daremo degli strumenti operativi per valutare l'efficienza di un generico algoritmo considerato.

Supponiamo di considerare un algoritmo A e di voler caratterizzare la sua complessità computazionale. Allora, se indichiamo con T il tempo impiegato dall'algoritmo, ed n la dimensione della struttura dati d su cui opera A, noi siamo interessati a caratterizzare la funzione:

$T=T(n)$.

A titolo di esempio se A è un programma che realizza l'ordinamento di un vettore, è chiaro che n è il riempimento del vettore che influenza , come noto, il tempo di esecuzione di A.

Spesso un algoritmo ha un tempo di esecuzione che dipende dalla dimensione di più strutture dati. Si pensi ad un programma che realizza la fusione di due vettori ordinati; in questo caso è evidente che la funzione T dipende sia dalla lunghezza del primo vettore che dalla lunghezza del secondo e quindi è una funzione di più variabili.

Nel corso del capitolo corrente il numero di variabili da cui dipende T non influenza i risultati e le considerazioni che tracceremo e, di conseguenza, per motivi di semplicità faremo riferimento ad una funzione T di una sola variabile.

Visto che è necessario prescindere dal particolare compilatore o dalla architettura del calcolatore su cui verrà eseguito l'algoritmo, il modello che si assumerà per il calcolo del tempo $T(n)$ è il modello computazionale tipicamente indicato con *random-access machine* (*RAM*). Nel modello RAM si considera un generico mono-processore in cui tutte le istruzioni sono eseguite una dopo l'altra, senza nessun tipo di parallelismo. Inoltre si assumerà che le istruzioni semplici del linguaggio (istruzioni di assegnamento, istruzioni con operatori aritmetici, relazionali o logici) abbiano un costo unitario in termini di tempo impiegato per la loro esecuzione. Su questa base si procederà al calcolo del tempo complessivo T impiegato da tutte le istruzioni dall'algoritmo, tenendo conto ovviamente del fatto che tale tempo dovrà dipendere dalla dimensione n dei dati, visto che siamo interessati alla valutazione di $T(n)$.

Un'altra considerazione importante che è qui opportuno fare, è che il tempo $T(n)$ potrebbe variare sensibilmente in dipendenza dello specifico valore assunto dalla struttura dati in ingresso.

Quello che si vuol dire è che T oltre a dipendere dalla dimensione n della struttura dati d, potrebbe dipendere anche dagli n valori assunti da una generica occorrenza di d. Sempre rifacendoci al caso di un algoritmo A che deve effettuare l'ordinamento di un vettore, potrebbe capitare che A impieghi un tempo sensibilmente diverso se il vettore in ingresso e già ordinato rispetto al caso in cui tale vettore sia non ordinato (o magari addirittura contrordinato).

Per questo motivo l'analisi che si conduce per il calcolo del tempo $T(n)$ va fatta esaminando tre casi possibili: il *caso migliore*, che corrisponde a quelle (o a quella) configurazioni della struttura dati d che danno luogo ad un minimo della funzione $T(n)$, il *caso peggiore* che corrisponde a quelle (o a quella) configurazioni della struttura dati d che corrispondono ad un massimo della funzione $T(n)$, ed il *caso medio*, che evidentemente corrisponde al comportamento medio della funzione al variare della configurazione di d. Dovrebbe risultare chiaro, ma conviene comunque precisarlo, che l'algoritmo considerato potrà anche avere un tempo di esecuzione $T(n)$ indipendente dai valori assunti da d. In tal caso il tempo impiegato nel caso migliore sarà uguale a quello impiegato nel caso peggiore, e quindi anche a quello impiegato nel caso medio.

Nel corso dei prossimi paragrafi, si analizzerà dapprima come è possibile caratterizzare con delle opportune notazioni il tempo di calcolo di un algoritmo A al crescere della dimensione n della struttura dati su cui A opera e si illustrerà il perché di questo tipo di analisi. In seguito si vedrà come è possibile utilizzare tali notazioni per caratterizzare la complessità computazionale dei principali costrutti di programmazione, ivi comprese le chiamate di funzioni.

Infine verrà affrontato il problema di calcolare il tempo di esecuzione di funzioni ricorsive: saranno introdotte delle opportune formule, dette *formule di ricorrenza*, e metodi per la loro soluzione.

Ordine di grandezza delle funzioni

Notazioni asintotiche

Nello studio della complessità di un algoritmo l'interesse è spesso quello di verificare il comportamento dell'algoritmo, in termini di tempo di calcolo, senza far riferimento ad una prefissata dimensione n dei dati di ingresso. In particolare, visto che l'incidenza sul tempo di calcolo cresce al crescere di n, l'analisi che si conduce è un analisi al limite, ovvero si studia il comportamento dell'algoritmo per un n sufficientemente grande. Le notazioni che si introducono sono pertanto dette *notazioni asintotiche*, in quanto devono caratterizzare il comportamento di un dato algoritmo a partire da un valore di n sufficientemente grande.

Un altro concetto da tener presente nella valutazione del comportamento di un algoritmo è il seguente: affermare che un algoritmo ha un certo andamento asintotico, significa dire che per *qualunque* configurazione di dati in ingresso il comportamento asintotico è di quel tipo; in altri termini non è corretto valutare il comportamento asintotico solo nel caso migliore (o in quello peggiore), ma è necessario verificare l'andamento dell'algoritmo in tutti i casi possibili. Sono tuttavia corrette affermazioni del tipo: *l'algoritmo A ha un comportamento asintotico, nel caso migliore, di tipo T*, anche se tale comportamento è diverso nel caso medio ed in quello peggiore.

Tre notazioni: motivazioni e definizioni

Nelle definizioni delle notazioni asintotiche ritroviamo formalizzato il concetto di analisi al limite: per dire che una funzione $f(n)$ appartiene ad un certo insieme $X(g(n))$ è necessario che l'andamento relativo di $f(n)$ e $g(n)$ sia di un certo tipo *a partire da una prefissata dimensione del dato di ingresso*, ovvero per ogni n maggiore di un certo n_0.

In particolare, in informatica, sono tre le notazioni comunemente adottate: O, W, Q. Una delle ragione per cui vengono adottate tre notazioni è che le prime due, come sarà evidenziato nel seguito, forniscono un limite "lasco", rispettivamente per i limiti superiore ed inferiore, mentre la

terza fornisce un limite "stretto". In alcuni contesti è difficile trovare un limite stretto per l'andamento delle funzioni, per cui ci si accontenta di un limite meno preciso.

Tali notazioni furono introdotte in un classico articolo di Knuth del '76; tuttavia in molti testi viene riportata una sola di queste notazioni, che in genere è la *O*. Tale scelta è dettata da ragioni di semplicità, sia per non introdurre troppe notazioni (cosa che potrebbe confondere le idee al lettore), sia perché in genere ciò che serve è una limitazione superiore del tempo impiegato da un dato algoritmo e, in quest'ottica, dimostrare che un algoritmo appartiene alla classe *O* è, come detto, più semplice che dimostrare l'appartenenza alla classe *Q*. Vi è inoltre un ultimo punto che vale la pena rimarcare: una notazione asintotica deve, ove possibile essere semplice, tant'è che, come vedremo, l'utilizzo di tali notazioni ci consente di trascurare costanti moltiplicative e termini di ordine inferiore. Orbene, spesso, volendo trovare un limite stretto, è necessario ricorrere a funzioni più complesse di quelle che si potrebbero adottare se ci si limitasse a considerare un limite lasco. Più in generale, se si vuole caratterizzare un algoritmo con un limite stretto può essere necessario dover considerare separatamente il caso migliore e quello peggiore, mentre se ci limita a cercare un limite superiore basta trovarlo per il solo caso peggiore ed evidentemente tale limite sarà valido per l'algoritmo stesso; quest'ultima considerazione può essere un'ulteriore giustifica all'adozione in alcuni testi di una sola notazione, la *O*.

Passiamo ora ad introdurre le definizioni delle tre notazioni asintotiche.

O(g(n))

Date due costanti positive c ed n_0, una funzione $f(n)$ appartiene all'insieme $O(g(n))$, ovvero $f(n) \in O(g(n))$ se:

$\exists\, c, n_0 > 0 \mid \forall\, n > n_0,\ 0 \le f(n) \le c\, g(n)$

ciò significa che, a partire da una certa dimensione n_0 del dato di ingresso, la funzione $g(n)$ maggiora la funzione $f(n)$. Possiamo quindi anche dire che la $g(n)$ rappresenta un limite superiore per la $f(n)$. Tale limite non è però "stretto". Supponiamo infatti di avere una funzione $f(n) \in O(n^2)$. Ciò implica che la $f(n)$, da un certo punto in poi, è maggiorata da n^2: se ciò è vero, anche n^3 maggiorerà la $f(n)$, e quindi $f(n)$ appartiene anche a $O(n^3)$. E' evidente che quest'ultima appartenenza implica un limite sicuramente meno stretto del precedente, ma rimane comunque formalmente ineccepibile. In generale quando si introduce la notazione *O* (soprattutto in quei contesti in cui è l'unica notazione presentata) si cerca comunque di individuare un limite superiore il più possibile stretto, fermo restando che tale limite potrebbe essere raffinato ulteriormente.

Si noti inoltre che il comportamento per tutti gli $n < n_0$ non è assolutamente tenuto in conto, per cui potranno esserci dei valori di $n < n_0$ tali che $f(n) > g(n)$, come evidenziato anche in Fig. 1b.

Ω(g(n))

Date due costanti positive c ed n_0 , una funzione $f(n)$ appartiene all'insieme $W(g(n))$, ovvero $f(n) \in W(g(n))$ se:

$\exists\, c, n_0 > 0 \mid \forall\, n > n_0,\ f(n) \ge c\, g(n) \ge 0$

ovvero, a partire da una certa dimensione n_0 del dato di ingresso, la funzione $g(n)$ è maggiorata dalla funzione $f(n)$. Anche in questo caso il limite non è "stretto", e valgono sostanzialmente tutte le considerazioni fatte per la notazione $O(n)$ (cfr. anche Fig. 1c).

Θ(g(n))

Date tre costanti positive c_1, c_2 ed n_0 , una funzione $f(n)$ appartiene all'insieme $Q(g(n))$, ovvero $f(n) \in Q(g(n))$ se:

$\exists\, c_1, c_2, n_0 > 0 \mid \forall\, n > n_0,\ 0 \le c_1\, g(n) \le f(n) \le c_2\, g(n)$

ovvero a partire da una certa dimensione n_0 del dato di ingresso, la funzione $f(n)$ è compresa tra $c_1\, g(n)$ e $c_2\, g(n)$. In maniera informale si può dire che, al crescere di n, la $f(n)$ e la $g(n)$ crescono "allo stesso modo" (vedi Fig. 1a). A differenza delle notazioni precedenti, dunque, se una funzione appartiene ad esempio a $Q(n^2)$, non potrà appartenere anche a $Q(n)$, né tantomeno a $Q(n^3)$

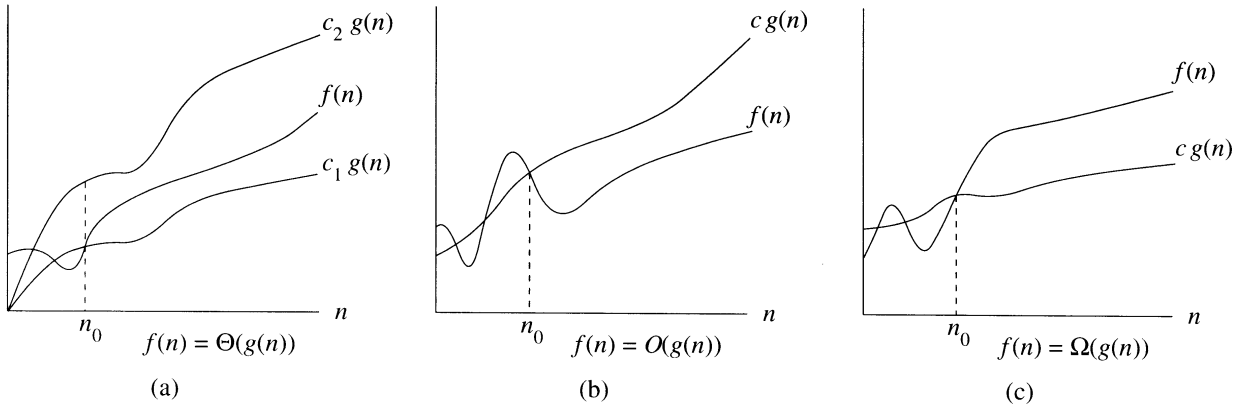

Fig. 1: Esempi di funzioni $f(n)$ che appartengono rispettivamente agli insiemi (**a**) $Q(g(n))$, (**b**) $O(g(n))$ e (**c**) $W(g(n))$. Si noti come le relazioni di diseguaglianza che compaiono nelle definizioni sono soddisfatte solo a partire da un certo valore n_0 della dimensione del dato di ingresso.

Implicazioni

Dalle definizioni suvviste discende il seguente teorema (che non dimostreremo):

> Date due funzioni $f(n)$ e $g(n)$, una funzione $f(n) \in Q(g(n))$ se e solo se $f(n) \in O(g(n))$ e $f(n) \in W(g(n))$.

Uso improprio

Come detto l'utilizzo corretto delle notazioni asintotiche è in espressioni del tipo $f(n) \in O(g(n))$. E' tuttavia prassi comune ammettere usi del tipo $f(n) = O(n^2)$. Inoltre in alcuni casi è molto utile dal punto di vista notazionale, anche se formalmente scorretto, poter sommare due notazioni asintotiche, cioè ammettere espressioni del tipo $T(n) = Q(n^2) + Q(n)$. Quest'ultima espressione deve evidentemente intendersi come: $T(n)$ è uguale alla somma *di una qualunque funzione che appartiene all'insieme* $Q(n^2)$ più *una qualunque funzione che appartiene all'insieme* $Q(n)$.

Alcune considerazioni

Come detto all'inizio del capitolo, non è corretto considerare solo il caso migliore per valutare il comportamento asintotico di un algoritmo. Solo in virtù del fatto che, ad es., il *BubbleSort* nel caso migliore ha una complessità lineare non basta per affermare che l'algoritmo *BubbleSort* appartenga a $Q(n)$ oppure a $O(n)$. Per lo stesso motivo non è corretto neppure affermare che il *BubbleSort* appartiene a $Q(n^2)$ solo perché nel caso peggiore la complessità è quadratica. Viceversa è corretto affermare che, nel *caso migliore*, la complessità del *BubbleSort* è $Q(n)$, che nel *caso medio* e nel *caso peggiore* la complessità è $Q(n^2)$ e, più in generale che il *BubbleSort* è $O(n^2)$ ovvero è $W(n)$.

Altre due importanti considerazioni fanno riferimento a casi in cui non sono verificate le condizioni che sono alla base dell'introduzione delle notazioni asintotiche. Infatti, proprio perché le notazioni introdotte sono asintotiche, vengono trascurati i termini di ordine inferiore e le costanti moltiplicative. Tuttavia, nel caso in cui è necessario confrontare algoritmi cui aventi tempi di esecuzione $T(n)$ il cui andamento al limite è uguale (es. sono entrambe $O(n^2)$), non è più possibile trascurare tali termini. In tal caso per stabilire quale algoritmo è più conveniente usare bisogna necessariamente tener conto, in primo luogo delle costanti moltiplicative, in primo luogo, e poi dei termini di ordine inferiore.

Un esempio classico è dato dalla scelta degli algoritmi di ordinamento: come dimostreremo nel seguito il *MergeSort* è un algoritmo che appartiene a $Q(n\log n)$, ovvero la sua complessità è sia nel caso migliore che in quello peggiore (e quindi anche in quello medio) $Q(n\log n)$. Il *QuickSort*,

viceversa è un algoritmo $O(n^2)$, la cui complessità nel caso migliore e nel caso medio è però $Q(nlogn)$. Contrariamente a quanto si potrebbe pensare, l'algoritmo di ordinamento più comunemente usato, e che si trova normalmente implementato nelle librerie, è proprio quest'ultimo. Questa scelta ha origine da due ordini di considerazioni: la prima è di carattere "tecnico", dovuta al fatto che il *QuickSort* svolge l'ordinamento sul posto, e che quindi ha minore complessità spaziale del *MergeSort*, la seconda, invece, è dovuta proprio al fatto che, da un lato la probabilità che per il *QuickSort* si verifichi il caso pessimo è abbastanza remota (e quindi considerare anche il *QuickSort* come un algoritmo di complessità $Q(nlogn)$, pur non essendo formalmente corretto, è una assunzione abbastanza prossima alla realtà) dall'altro che le costanti moltiplicative nel caso del *QuickSort* sono minori rispetto a quelle del *MergeSort*. Quindi per scegliere tra due algoritmi che hanno "sostanzialmente" lo stesso ordine di complessità è necessario considerare il peso delle costanti moltiplicative e queste giocano a favore del *QuickSort*.

Un altro caso in cui non è corretto trascurare i termini "nascosti" dalla notazione asintotica è il caso in cui siamo interessati a confrontare il comportamento di due algoritmi per un *prefissato n*. In tal caso è possibile, ad esempio, che un algoritmo A_1 di complessità $Q(n^3)$ si comporti meglio di un algoritmo $A_2 \in Q(n^2)$; per convincersene supponiamo di aver fissato $n = 50$ e che l'algoritmo A_1 abbia un tempo $T(n)$ dato da $n^3 / 10$ mentre l'algoritmo A_2 abbia $T(n)$ uguale a $10n^2 + 2n + 10$. In tal caso per A_1 avremo $T(50) = 12500$, mentre per A_2 avremo $T(50) = 25110$.

Calcolo della complessità computazionale dei principali costrutti di programmazione

Nel seguito daremo delle semplici regole per caratterizzare la complessità dei vari costrutti di programmazione, sulla base delle notazioni asintotiche introdotte nei precedenti paragrafi. Combinando opportunamente tali regole è possibile calcolare la complessità computazionale di un generico algoritmo.

Istruzioni

Una volta fissato il modello di costo della macchina RAM, le istruzioni di assegnamento, le istruzioni che coinvolgono solo operatori aritmetici, relazionali o logici hanno tutte, come detto, lo stesso costo computazionale. In particolare si assume che tali istruzioni abbiano complessità costante, per indicare la quale viene convenzionalmente utilizzata la notazione $Q(1)$ (o equivalentemente $O(1)$).

Blocchi

La complessità di un blocco sequenziale di istruzioni è data dalla complessità massima fra quelle delle istruzioni costituenti il blocco in questione. Dette $I_1,\ldots,I_i,\ldots,I_m$ le m istruzioni del blocco e $f_1(n),\ldots,f_i(n),\ldots,f_m(n)$, le funzioni che ne caratterizzano la complessità, la complessità del blocco sarà quindi data da: $Q(\max_i(f_i(n)))$.

In questo caso si applica la cosiddetta *regola della somma:* cioè è possibile calcolare la complessità di una sequenza di istruzioni come la somma delle complessità delle istruzioni componenti e la complessità risultante è quella del termine di grado più alto. Se ad esempio ho un blocco di tre istruzioni, la cui complessità è rispettivamente $Q(1)$, $Q(n)$ e $Q(1)$, posso calcolare la complessità del blocco (indicandola con Q_{blocco}) come: $Q_{blocco} = Q(1) + Q(n) + Q(1)$, che per la regola della somma sarà uguale a $Q(n)$.

Costrutti selettivi

La complessità di un istruzione del tipo *if* <condizione> *then* <istruzioni ramo then> *else* <istruzioni ramo else> è limitata superiormente dal ramo che ha la complessità maggiore. In realtà bisogna anche considerare il tempo impiegato a verificare la condizione, che in genere ha

complessità costante. Indicando per semplicità con f_{cond}, f_{else} ed f_{then} queste tre complessità e utilizzando la notazione O avremo che la complessità dell'*if* è pari a $O(\max(f_{cond}+f_{then}, f_{cond}+f_{else}))$, dove il simbolo + sta ad indicare che si applica la regola della somma.

Volendo invece utilizzare la notazione Q, siamo costretti, come accennato nell'introduzione, a distinguere un caso migliore ed un caso peggiore. Nel caso migliore avremo che la complessità sarà data da: $Q(\min(f_{cond}+f_{then}, f_{cond}+f_{else}))$, mentre nel caso peggiore sarà data da $Q(\max(f_{cond}+f_{then}, f_{cond}+f_{else}))$.

Ragionamenti analoghi valgono nel caso di un costrutto di tipo *case*.

Cicli

Ciclo While

La complessità di un ciclo *while* è in generale data dal prodotto della complessità del corpo del *while* stesso per il numero di volte che esso viene eseguito. In più bisogna considerare il tempo necessario alla verifica della condizione di fine ciclo. Trattandosi di ciclo non predeterminato, sarà necessario distinguere un caso migliore ed un caso peggiore. Dette k_{min} e k_{max} il numero minimo e massimo di volte che viene iterato un generico ciclo *while*, e dette f_{cond} e f_{corpo} le funzioni che caratterizzano le complessità derivanti rispettivamente dalla valutazione della condizione e dalle istruzioni che compongono il corpo del *while*, avremo che, nel caso migliore, la complessità sarà data da $Q(f_{cond} + k_{min} \cdot f_{corpo}))$, mentre nel caso peggiore sarà pari a $Q(f_{cond} + k_{max} \cdot f_{corpo}))$,. Si noti che è necessario considerare anche f_{cond} visto che k_{min} (e anche k_{max}…) potrebbe essere pari a 0. Utilizzando la notazione O anche in questo caso è possibile avere un'unica espressione per esprimere la complessità del ciclo *while*, che sarà infatti data da $O(f_{cond} + k_{max} \cdot f_{corpo}))$.

Ciclo For

La complessità di un ciclo *for* è data dal prodotto della complessità del corpo del *for* stesso per il numero di volte che esso viene eseguito. Anche in questo caso bisogna tener conto del fatto che è necessario valutare il tempo necessario all'inizializzazione della variabile di ciclo, al confronto con la condizione di fine ciclo e all'incremento della variabile stessa, tempo che tipicamente è costante. Detta quindi genericamente f_{cond} la funzione che caratterizza tali tempi, detta invece f_{corpo} la funzione che caratterizza le complessità derivante dalle istruzioni che compongono il corpo del *for* ed indicate con k il numero di iterazioni, si avrà che la complessità del *for* è data da $Q(f_{cond} + k \cdot f_{corpo})$. Si noti tuttavia che, dal momento che in C il *for* di fatto non implementa un ciclo predeterminato, nella pratica valgono le stesse considerazioni fatte nel caso del *while*.

Chiamate di funzioni

La complessità di una chiamata di funzione è data dalla somma di due termini: il costo, in termini di complessità computazionale, dell'esecuzione della funzione stessa ed il costo della chiamata. Quest'ultimo è in genere trascurabile, ma non lo è se si effettuano chiamate per valore che hanno l'effetto di copiare sullo stack la struttura dati d la cui dimensione n è proprio quella da cui dipende la nostra funzione $T(n)$ di interesse. Tuttavia, se la struttura dati è un vettore ed il linguaggio considerato è il C, poiché alla funzione viene passato di fatto il puntatore al primo elemento del vettore, il tempo di chiamata è indipendente da n.

Calcolo della complessità di funzioni ricorsive

Nel caso in cui la funzione di cui si vuole calcolare la complessità è una funzione ricorsiva, cioè contiene al suo interno una chiamata a se stessa, la tecnica proposta nei precedenti paragrafi non può essere semplicemente applicata. Infatti, data una generica funzione ricorsiva R, la complessità

della chiamata di un'istanza ricorsiva di *R*, che compare nel corpo di *R* stessa, dovrebbe essere data dalla somma del costo di chiamata e del costo dell'esecuzione dell'istanza di *R*. Ma quest'ultimo costo è proprio quello che stiamo cercando di calcolare, ed è quindi incognito. Per risolvere questo problema è necessario esprimere il tempo incognito $T(n)$ dell'esecuzione di *R*, come somma di due contributi: un tempo $Q(f(n))$ che deriva dall'insieme di tutte le istruzioni che non contengono chiamate ricorsive, ed un tempo $T(k)$ che deriva dalle chiamate ricorsive, invocate su di una dimensione del dato di ingresso più piccola, cioè con $k < n$. Otterremo quindi un'equazione, detta *equazione ricorrente* o *formula di ricorrenza* o più semplicemente *ricorrenza*, del tipo $T(n)=aT(n/b)+Q(f(n)$ (oppure $T(n)=aT(n-b)+Q(f(n))$, dove il primo contributo è appunto quello di a chiamate ricorsive su di un dato di dimensione $1/b$ (oppure di b unità più piccolo) rispetto a quello di partenza ed il secondo contributo è quello legato a tutte le istruzioni della funzione in cui non compaiono chiamate ricorsive. Per completare la ricorrenza (e permetterne la risoluzione) è necessario anche valutare il tempo impiegato dalla funzione per valori di n sufficientemente piccoli. In questi casi (tipicamente per $n=0$ o $n=1$) la funzione termina senza attivare chiamate ricorsive e se ne può quindi calcolare semplicemente il tempo di esecuzione $T(n)$, che sarà in generale costante.

Ad esempio, nell'ambito del paradigma *divide-et-impera*, avremo ricorrenze del tipo:

$$T(n)=\begin{cases} Q(1) & per\ n \le c \\ aT(n/b)+C(n)+D(n) & per\ n > c \end{cases}$$, derivanti dall'aver diviso un certo problema padre in a sottoproblemi figlio di dimensione n/b, avendo indicato con $C(n)$ il costo derivante dalla combinazione dei risultati e con $D(n)$ il costo relativo alla divisione del problema padre nei problemi figlio ed essendo costante (e quindi pari a $Q(1)$) il tempo per la soluzione dei problemi nel caso banale, cioè quando $n \le c$.

Per risolvere una formula di ricorrenza, e quindi giungere a determinare la complessità asintotica di $T(n)$, è necessario adottare dei particolari metodi di soluzione che saranno presentati e discussi nel prossimo paragrafo.

Formule di ricorrenza: Metodi di soluzione

I metodi proposti per la soluzione delle formule di ricorrenza sono sostanzialmente tre: il metodo iterativo, il metodo di sostituzione ed il metodo principale.

Il primo metodo consiste semplicemente nell'iterare la ricorrenza proposta, finché non si riesce a trovare una generalizzazione. A questo punto occorre trovare il valore per il quale si chiude la ricorrenza, sostituirlo nella formula generale e calcolare il risultato applicando le regole per limitare le sommatorie. In generale è quindi necessario fare un po' di passaggi matematici (che non dovrebbero essere comunque particolarmente complessi).

Il metodo di sostituzione consiste nell'ipotizzare una soluzione candidata e dimostrare l'esattezza dell'ipotesi sulla base dell'induzione matematica. Il problema si riconduce dunque alla scelta della funzione candidata. Nel caso in cui la funzione scelta non si dimostri corretta, sarà necessario provare con un'altra funzione.

Il metodo principale, viceversa, permette di trovare direttamente la soluzione in un certo numero di casi. In particolare questo metodo si può applicare a funzioni la cui formula di ricorrenza sia del tipo $T(n)=aT(n/b)+f(n)$, cioè il problema padre è divisibile in a problemi figlio tutti di dimensioni $1/b$ rispetto al padre, ed $f(n)$ è il costo della fase di divisione e combinazione.

In tal caso occorre confrontare la funzione $f(n)$ con la funzione $n^{\log_b a}$: la complessità risultante sarà quella della funzione di ordine maggiore; se le due funzioni sono dello stesso ordine comparirà un termine logaritmico nella soluzione dell'equazione di ricorrenza.

Più precisamente, nel caso in cui $f(n) \in Q(n^{\log_b a})$, cioè $f(n)$ e $n^{\log_b a}$ sono dello stesso ordine di grandezza, la soluzione dell'equazione di ricorrenza sarà $Q(f(n) \log n)$; se viceversa $f(n) \in O(n^{\log_b a-e})$ per qualche $\varepsilon > 0$ (questo equivale a dire che la funzione $f(n)$ è maggiorata *polinomialmente* da $n^{\log_b a}$) la complessità risultante sarà data ovviamente dalla maggiore delle due

funzioni, cioè la soluzione della ricorrenza sarà $Q(n^{\log_b a})$. L'ultimo caso è quello in cui si verifica che $f(n) \in W(n^{\log_b a+e})$ per qualche $\varepsilon > 0$ e in più è anche verificata una condizione di "regolarità", cioè $a\, f(n/b) \pounds c\, f(n)$ per qualche $c < 1$. In tal caso la funzione $f(n)$ maggiora polinomialmente $n^{\log_b a}$ e quindi la soluzione sarà $Q(f(n))$.

Esempi

Calcolo del fattoriale

```
int fattoriale(int n){

 if (n==1)
   return 1;
 else return n*fattoriale(n-1);
}
```

La formula di ricorrenza è data in questo caso da: $T(n) = \begin{cases} Q(1) & per\ n = 1 \\ T(n-1) + Q(1) & per\ n > 1 \end{cases}$. Infatti nel caso base, quando n==1, la complessità è $Q(1)$ perché bisogna solo effettuare un confronto e poi restituire 1. Nel passo induttivo, si spende un tempo $Q(1)$ per la verifica della condizione dell'*if* e ancora $Q(1)$ per il prodotto e la restituzione del risultato, mentre la chiamata viene effettuata su una dimensione che è di un'unità più piccola: quindi la complessità sarà data da $T(n-1) + Q(1)$. Risolviamo la ricorrenza con il metodo iterativo:

$T(n) = T(n-1) + Q(1) = (T(n-2) + Q(1)) + Q(1) = ((T(n-3) + Q(1)) + Q(1)) + Q(1)$;

da cui generalizzando, otteniamo:

$T(n) = T(n-k) + \sum_{i=1}^{k} Q(1)$.

La ricorrenza si chiude quando $k = n-1$. In questo caso infatti l'argomento di $T(\cdot)$ sarà pari ad uno ed arriveremo al caso banale $T(1) = Q(1)$. Quindi sostituendo otteniamo:

$$T(n) = T(1) + \sum_{i=1}^{n-1} Q(1) = Q(1) + \sum_{i=1}^{n-1} Q(1) = \sum_{i=1}^{n} Q(1) = Q(n).$$

La Torre di Hanoi

```
typedef int piolo;

void muovi(int disco, piolo sorgente, piolo destinazione) {

  printf("Muovi il disco %2d da %2d a %2d\n", disco, sorgente, destinazione);
}


void hanoi(int n, piolo sorgente, piolo destinazione, piolo ausiliario) {

  if (n == 1)
    muovi(1, sorgente, destinazione);
  else {
    hanoi(n-1, sorgente, ausiliario, destinazione);
    muovi(n, sorgente, destinazione);
    hanoi(n-1, ausiliario, destinazione, sorgente);
  }
}
```

La funzione `muovi()` ha complessità $Q(1)$, essendo composta da un'unica istruzione di stampa. Allora la formula di ricorrenza sarà data da: $T(n)=\begin{cases} Q(1) & per\ n=1 \\ 2T(n-1)+Q(1) & per\ n>1 \end{cases}$. Infatti nel caso banale devo effettuare solo un confronto e la chiamata di `muovi()`, che hanno entrambe complessità $Q(1)$. Nel passo induttivo ho invece due chiamate ricorsive, su di un argomento di un'unità più piccolo (che costano quindi $2T(n-1)$) ed una chiamata di `muovi()` che ha complessità $Q(1)$. Posso ancora risolvere la ricorrenza con il metodo iterativo:

$T(n) = 2T(n-1) + Q(1) = 2(2T(n-2) + Q(1)) + Q(1) = 2(2(2T(n-3) + Q(1)) + Q(1))+ Q(1)$;

effettuando le moltiplicazioni ottengo:

$T(n) = 8T(n-3) + 4Q(1) + 2Q(1) + Q(1)$

da cui generalizzando:

$$T(n) = 2^k\,T(n-k) + \sum_{i=0}^{k-1} 2^i Q(1) = 2^k\,T(n-k) + Q\left(\sum_{i=0}^{k-1} 2^i\right).$$

Ancora una volta la ricorrenza si chiude quando $k = n-1$. In questo caso infatti l'argomento di $T(\cdot)$ sarà pari ad uno ed arriveremo al caso banale $T(1)= Q(1)$. Quindi sostituendo otteniamo:

$$T(n) = 2^{n-1}\,T(1) + Q\left(\sum_{i=0}^{n-2} 2^i\right) = 2^{n-1}Q(1) + Q\left(\sum_{i=0}^{n-2} 2^i\right) = Q\left(\sum_{i=0}^{n-1} 2^i\right).$$

La sommatoria in parentesi non è altro che la serie geometrica. In genere:

$\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}$. Nel nostro caso allora avremo: $\sum_{i=0}^{n-1} 2^i = \frac{2^n-1}{2-1} = 2^n - 1$, da cui sostituendo nella relazione precedentemente trovata risulta:

$T(n) = Q(2^n - 1) = Q(2^n)$.

MergeSort

```
#define MAX_SIZE 1000
typedef int Vettore[MAX_SIZE];

void Merge(Vettore vet, int i, int j, int k) {
  Vettore aux;
  int p, p1 = i, p2 = j + 1;

  for (p = i; p <=k; p++)
    if (p1 > j)
      aux[p] = vet[p2++];
    else if (p2 > k)
           aux[p] = vet[p1++];
         else if (vet[p1] <= vet[p2])
                aux[p] = vet[p1++];
              else
                aux[p] = vet[p2++];

  for (p = i; p <= k; p++)
    vet[p] = aux[p];
}
```

La complessità della funzione `Merge()` è data dalla somma della complessità delle inizializzazioni e delle complessità dei due cicli *for*. Questi ultimi hanno entrambi un corpo di complessità $Q(1)$ che viene eseguito n volte. La complessità della `Merge()` è dunque $Q(1) + Q(n) + Q(n)$ che per la regola della somma risulta essere uguale a $Q(n)$.

```
void MergeSort(Vettore vet, int i, int j) {
  if (i < j) {
    MergeSort(vet, i, (i+j)/2);
    MergeSort(vet, (i+j)/2 + 1, j);
    Merge(vet, i, (i+j)/2, j);
  }
}
```

La formula di ricorrenza in questo caso è data da: $T(n) = \begin{cases} Q(1) & per\ n = 1 \\ 2T(n/2) + Q(n) & per\ n > 1 \end{cases}$.

Infatti nel caso banale (vettore vuoto o composto da un unico elemento) devo effettuare solo un confronto che ha chiaramente complessità $Q(1)$. Nel passo induttivo invece devo effettuare un confronto, con costo $Q(1)$, più due chiamate ricorsive su di un vettore di dimensione metà, che costano $2T(n/2)$, più una chiamata di `Merge()` sull'intero vettore, chiamata che ha quindi complessità $Q(n)$ (che in questo caso rappresenta il costo di combinazione indicato con $C(n)$ nel precedente paragrafo, mentre non ho un costo di divisione $D(n)$). Di nuovo, per la regola della somma, $Q(1) + Q(n) = Q(n)$.

Risolviamo dapprima la ricorrenza con il metodo iterativo.

$T(n) = 2T(n/2) + Q(n) = 2(2T(n/4) + Q(n/2)) + Q(n) = 2(2(2T(n/8)+ Q(n/4)) + Q(n/2)) + Q(n)$;

effettuando le moltiplicazioni ottengo:

$T(n) = 8T(n/8) + Q(n) + Q(n) + Q(n)$

da cui generalizzando:

$T(n) = 2^k\, T(n/2^k) + \sum_{i=1}^{k} Q(n)$.

Ancora una volta la ricorrenza si chiude quando l'argomento di $T(\cdot)$ è pari ad 1; in questo caso ciò si verifica quando $n=2^k$. Ciò implica che k sarà uguale a $\log_2 n$. Se sostituiamo nella precedente relazione avremo allora:

$T(n) = n\, T(1) + \sum_{i=1}^{\log n} Q(n) = nQ(1) + \sum_{i=1}^{\log n} Q(n) = Q(n) + Q(n \log n) = Q(n \log n)$.

A questo risultato era possibile anche giungere direttamente applicando il metodo principale. Nel nostro caso infatti, la relazione di ricorrenza è del tipo $T(n)=aT(n/b)+f(n)$, con $a = 2$, $b = 2$ e $f(n) = Q(n)$. Confrontando allora $f(n) = Q(n)$ con $n^{\log_b a} = n^{\log_2 2} = n$, troviamo che $f(n) \in Q(n^{\log_b a})$, pertanto la soluzione dell'equazione di ricorrenza sarà $Q(f(n) \log n) = Q(n \log n)$.

Proviamo ora ad applicare anche il metodo di sostituzione, utilizzando chiaramente la funzione $n \log n$ come funzione candidata.

Per semplicità dimostriamo solo che il `MergeSort()` è $O(n \log n)$. In effetti per dimostrare l'appartenenza a $Q(n \log n)$, dovremmo dimostrare anche che il `MergeSort()` è $W(n \log n)$, dimostrazione che, per brevità, sarà omessa. Conduciamo la dimostrazione per induzione. Partiamo dal passo induttivo: dobbiamo dimostrare che $T(n) \le c\, n \log n$, supponendo per induzione completa che $T(n/2) \le c\, (n/2) \log(n/2)$.

Sostituendo l'ipotesi induttiva nella ricorrenza $T(n)=2T(n/2) + n$, otteniamo:

$T(n) \le 2\, c\, (n/2) \log(n/2) + n = cn\, (\log n - \log 2) + n = cn \log n - cn + n \le cn \log n$.

L'ultimo passaggio è evidentemente valido per $c \ge 1$.

Per completare la dimostrazione dobbiamo dimostrare anche la base, cioè che, per un certo k, sia $T(k) \le c\, k \log k$. Normalmente come valore di k viene scelto 0 oppure 1. In questo caso tuttavia, scegliere k pari ad 1 porta a dover dimostrare la diseguaglianza $T(1) \le c\ 1 \log 1$, che equivale a dover dimostrare $T(1) \le 0$, il che è impossibile. Il principio d'induzione, tuttavia, dice che, dimostrato il passo induttivo, basta dimostrare il caso base per un certo valore di k per garantire la verità dell'asserto per tutti gli $n \ge k$. Dimostrare quindi l'asserto $T(k) \le c\, k \log k$ per $k = 2$ garantisce che tale asserto sia vero per tutti gli $n \ge 2$. Ciò è più che sufficiente, dal momento siamo interessati a

dimostrare la veridicità di una notazione asintotica. Dobbiamo quindi dimostrare che $T(2) \leq c\ 2\log 2$. Il primo membro, applicando la definizione induttiva è pari a 4; otteniamo pertanto $4 \leq 2c$, che risulta evidentemente vero per ogni $c \geq 2$.

QuickSort

```
int Partition(int *vet,int i,int j){
  int first,last,pivot;

  pivot = vet[i];
  first = i-1;
  last = j+1;
  for(;;){
    do last--;
     while (vet[last]>pivot);
    do first++;
     while (vet[first]<pivot);
    if (first<last)
     swap(vet,first,last);
    else return last;
  }
}
```

La complessità della funzione `Partition()` è data dalla somma della complessità delle inizializzazioni e della complessità del ciclo *for*. Quest'ultimo ha una complessità $Q(n)$ in quanto l'istruzione di `return`, che determina la fine del ciclo viene raggiunta dopo l'esecuzione di un numero di istruzioni dell'ordine di n.

```
void QuickSort(int *vet, int i,int j){
  int k;

  if (i < j){
    k = Partition(vet,i,j);
    QuickSort(vet,i,k);
    QuickSort(vet,k+1,j);
  }
}
```

La formula di ricorrenza in questo caso dipende da come lavora la funzione `Partition()`, il cui costo rappresenta il costo di divisione che era stato indicato con $D(n)$ nel precedente paragrafo, mentre non c'è un costo di combinazione $C(n)$. Nel caso in cui l'indice `k` restituito da `Partition()` è tale da dividere il vettore in due sottovettori di dimensione metà, la formula di ricorrenza è evidentemente la stessa trovata per il `MergeSort`, cioè: $T(n) = \begin{cases} Q(1) & per\ n = 1 \\ 2T(n/2) + Q(n) & per\ n > 1 \end{cases}$. Infatti nel caso banale (vettore vuoto o composto da un unico elemento) devo effettuare solo un confronto che ha chiaramente complessità $Q(1)$. Nel passo induttivo invece devo effettuare una chiamata di `Partition()` sull'intero vettore, con costo $Q(n)$, e due chiamate ricorsive, su di un vettore di dimensione metà, che costano $2T(n/2)$. In questo caso, che è anche il *caso migliore*, poiché la formula di ricorrenza è la stessa del `MergeSort`, la soluzione sarà banalmente $Q(n \log n)$.

Viceversa, se l'indice `k` restituito da `Partition()`è tale da dividere il vettore in due sottovettori, uno di dimensione unitaria e l'altro di dimensione n–1, la formula di ricorrenza

diventerà: $T(n)=\begin{cases} Q(1) & per\ n=1 \\ T(n-1)+Q(n) & per\ n>1 \end{cases}$. Infatti nel passo induttivo, la complessità sarà data dalla somma della complessità di `Partition()` che è $Q(n)$ più la complessità delle due chiamate ricorsive, che avranno in questo caso complessità $T(1)$ e $T(n–1)$. Poiché $T(1) = Q(1)$, avremo che nel passo induttivo, la complessità sarà data da $Q(n) + Q(1) + T(n–1)$, che, sempre per la regola della somma, è pari a $T(n–1) + Q(n)$.

Risolviamo allora la ricorrenza con il metodo iterativo:

$T(n) = T(n–1) + Q(n) = (T(n–2) + Q(n–1)) + Q(n) = ((T(n–3) + Q(n–2)) + Q(n–1)) + Q(n)$;

da cui generalizzando:

$T(n) = T(n–k) + \sum_{i=1}^{k} Q(n-i+1)$.

La ricorrenza si chiude quando $k = n–1$. In questo caso infatti l'argomento di $T(\cdot)$ sarà pari ad uno ed arriveremo al caso banale T(1)= $Q(1)$. Quindi sostituendo otteniamo:

$$T(n) = T(1) + \sum_{i=1}^{n-1} Q(n-i+1) = Q(1) + \sum_{i=1}^{n-1} Q(n-i+1) = Q\left(\sum_{i=1}^{n}(n-i+1)\right) = Q(n^2).$$

L'ultimo passaggio deriva dal fatto che tra parentesi ho una serie aritmetica, la cui sommatoria è pari a $\frac{1}{2}n(n+1)$.

Il caso appena considerato corrisponde al *caso peggiore* per l'algoritmo `QuickSort`. Si può dimostrare che nel *caso medio* la complessità del `QuickSort` è uguale a quella del caso migliore, cioè è $Q(n\log n)$.