Teoria della complessità

Introduzione alla complessità degli algoritmi

Donato Malerba

Premessa

- Lo studio della computabilità aiuta a comprendere quali problemi ammettono una soluzione algoritmica e quali no.
- Per i problemi computabili, è interessante conoscere la complessità degli algoritmi che li risolvono.
- Algoritmo: sequenza di passi computazionali che trasformano un valore (o un insieme di valori) in ingresso in un valore (o un insieme di valori) in uscita.

Premessa

- La *complessità di un algoritmo* corrisponde a una misura delle risorse di calcolo consumate durante la computazione.
- È tanto più elevata quanto maggiori sono le risorse consumate.

Teoria della complessità

- *Teoria della complessità computazionale*: branca dell'informatica che studia l'uso delle risorse di calcolo necessarie per la computazione di algoritmi e per la soluzione di problemi.
- Questo studio è importante al fine di:
  – poter confrontare algoritmi diversi che risolvono lo stesso problema
  – stabilire se dei problemi ammettono algoritmi risolutivi che non sono *praticamente* computabili, cioè che richiedono risorse che non sono praticamente disponibili.

Misure statiche

- Le misure statiche sono basate sulle caratteristiche dell'algoritmo e prescindono dai valori dell'input su cui esso opera.

Esempi:

Numero di istruzioni

Numero di cicli → *misura strutturale*

Misure dinamiche

- Le risorse di interesse predominante nella teoria della complessità sono:
- *tempo* → *complessità in tempo*
- *spazio di memoria* → *complessità in spazio*

Queste sono misure dinamiche, che tengono conto sia delle caratteristiche dell'algoritmo che dell'input su cui esso opera.

Sono misure basate sulle caratteristiche di una computazione.

Definizione assiomatica

Una misura dinamica Φ è un insieme di funzioni calcolabili Φ_i associate agli algoritmi P_i tale che:

1. Per ogni *i*, $P_i(x)$ termina se e solo se $\Phi_i(x)$ è definita;
2. Esiste un predicato totale calcolabile M tale che M(i,x,m)=vero se e solo se $\Phi_i(x)=m$.

M. Blum (1967). A machine independent theory of the complexity of recursive functions, *J. of the ACM*, 14, 322-336.

Definizione assiomatica

La secondo assioma assicura che esiste un modo effettivo per decidere se una computazione ha speso non più di *m* unità di risorsa.

Utilizzando ripetutamente il predicato M risulta quindi possibile calcolare, per qualunque computazione $P_i(x)$ che termina, la misura dinamica di complessità $\Phi_i(x)$ associata a $P_i(x)$.

Definizione assiomatica

Si osservi che il primo assioma *non* implica il secondo.

Se prendiamo come Φ_i gli stessi algoritmi P_i il primo assioma sarebbe verificato.

Per assurdo supponiamo che anche il secondo assioma sia verificato. Se supponiamo che gli algoritmi P_i siano descritti da matrici funzionali di macchine di Turing T_i, dev'essere *decidibile* il predicato $T_i(x)=y$. Ma questo implicherebbe la possibilità di risolvere il problema della terminazione per macchine di Turing che è noto non essere decidibile.

Definizione assiomatica

Si osservi che il secondo assioma *non* implica il primo.

Se prendiamo $\Phi_i(x)=k$ costante per ogni *i*. In tal caso il secondo assioma è evidentemente verificato mentre non lo è il primo.

Se supponiamo che gli algoritmi P_i siano descritti da matrici funzionali di macchine di Turing T_i, la seguente definizione:

$\Phi_i(x)$= numero di transizioni eseguite da $T_i(x)$

soddisfa entrambi gli assiomi.

Definizione assiomatica

Blum trasse da questa definizione delle proprietà delle misure di complessità del tutto indipendenti dal modello di calcolo.

Tuttavia, per caratterizzare sufficientemente la complessità di algoritmi e/o problemi occorre introdurre un *modello di calcolo* all'interno del quale definire particolari misure di complessità.

Misure dinamiche di complessità per macchine di Turing deterministiche

Come già visto, una misura dinamica della complessità in tempo per le macchine deterministiche può essere rappresentata dal numero di transizioni che occorrono in una computazione massimale.

Una misura dinamica della complessità in spazio potrebbe essere il numero di celle del nastro utilizzate da una macchina di Turing deterministica durante la sua computazione.

Misure dinamiche di complessità per macchine di Turing deterministiche

Potremmo raffinare questa definizione facendo riferimento allo spazio di lavoro *addizionale*, cioè alla quantità di celle di nastro utilizzate per eseguire una computazione, non considerando lo spazio richiesto per la descrizione dei dati di ingresso.

Si noti tuttavia che in ogni caso questa misura dinamica di complessità in spazio non soddisfa il primo assioma di misura dinamica, poiché una computazione può non terminare pur richiedendo una quantità finita di nastro.

Misure dinamiche di complessità per macchine di Turing deterministiche

Questo problema è aggirato considerando la misura dinamica di complessità in spazio non definita ogniqualvolta la computazione $T_i(x)$ non termina.

È possibile però sapere se una computazione $T_i(x)$ che utilizza un numero limitato di celle termina o meno? In altre parole, la definizione di misura dinamica di complessità in spazio è operativa, cioè soddisfa *entrambi* gli assiomi?

Misure dinamiche di complessità per macchine di Turing deterministiche

Lemma. *Se è possibile determinare un limite finito* k *sul numero di celle di nastro richiesto dalla computazione* $T_i(x)$*, allora è anche possibile decidere in tempo finito se la computazione termina o meno.*

Dimostrazione. Supponiamo che T_i sia una MT con un alfabeto di *c* simboli (incluso blank) ed *s* stati (s=|Q|). Per ipotesi esiste un limite superiore *k* sul numero di celle di nastro utilizzate nella computazione di $T_i(x)$.

Misure dinamiche di complessità per macchine di Turing deterministiche

Facciamo partire la macchina e osserviamola mentre esegue le prime $N=c^k sk$ transizioni. Se la macchina non si ferma prima, allora possiamo asserire che non si fermerà più perché è entrata in un ciclo. Infatti il valore di N è un limite superiore al numero di differenti configurazioni istantanee assumibili durante la computazione di $T_i(x)$: esso prevede che, per tutte le configurazioni di *c* simboli in *k* celle, la computazione scandisca tutte le *k* celle, e per ognuna di esse, passi in tutti gli *s* stati possibili.

Misure dinamiche di complessità per macchine di Turing non deterministiche

Nel caso di macchina di Turing non deterministica MTN_i la complessità in tempo è definita nel modo seguente.

Sia $MTN_i(x)$ una computazione non deterministica. Distinguiamo i tre casi:

1. Esiste almeno una computazione massimale accettante. Allora la complessità in tempo è definita come la minima complessità fra quelle massimali accettanti.
2. Tutte le computazioni sono massimali e rifiutanti. Allora la complessità in tempo è definita come la minima complessità fra quelle massimali rifiutanti.
3. Le computazioni massimali sono rifiutanti e almeno una non è massimale. Allora la complessità in tempo non è definita.

Misure dinamiche di complessità per macchine di Turing non deterministiche

La complessità in spazio nel caso di macchine di Turing non deterministiche è definita in modo analogo.

Misure dinamiche di complessità per macchine a registi

Le misure dinamiche di complessità per macchine a registri sono già state introdotte per studiare il costo di una simulazione di una RAM mediante una MT (e viceversa).

Una prima misura di complessità in tempo si limita a contare il numero di istruzioni eseguite in una computazione. Essa si basa sull'assunzione che ogni istruzione ha un costo unitario (*modello di costo uniforme*).

Misure dinamiche di complessità per macchine a registi

Una misura di complessità più realistica tiene conto della "taglia" degli operandi delle istruzioni (*modello a costi logaritmici*).

Analogamente possiamo definire una misura dinamica della complessità in spazio per RAM in due modi alternativi:

1. Numero di registri usati durante la computazione (assunzione di costo uniforme)
2. Somma su tutti i registri, incluso l'accumulatore, di $l(x_i)$, dove x_i è il massimo intero memorizzato nel registro i-esimo durante la computazione (assunzione di costo logaritmico)

Un modello di calcolo basato su linguaggio imperativo ad alto livello

Descriveremo un algoritmo mediante

- operazioni semplici, come assegnazione ed operazioni aritmetiche,
- operazioni composte di selezione (**if then else**)
- e iterazione (**while … do** , **repeat** … **until**).
- chiamate di procedure e funzioni

Pascal semplificato

Modello di costo

Per poter valutare la complessità in tempo o spazio di un algoritmo è necessario innanzitutto definire un *modello di costo* che stabilisca come calcolare il consumo di una risorsa.

Modello di costo temporale

Assumiamo quanto segue:

1. il costo di un'operazione di assegnazione, aritmetica, logica e di confronto sia unitario;
2. il costo di un'operazione di selezione

   if *cond* **then** *S1* **else** *S2*

   sia dato dal costo di esecuzione del test *cond* più la somma dei costi delle istruzioni del blocco S1, nel caso in cui la condizione sia vera, o la somma dei costi delle istruzioni del blocco S2 nel caso in cui la condizione sia falsa;

Modello di costo temporale

3. il costo di un'istruzione di ciclo

   while *cond* **do** *S1*

   sia dato da:

   costo del test *cond* + (costo del test *cond* + costo delle istruzioni in *S1*) * numero di volte in cui il ciclo è ripetuto;
4. il costo di un'istruzione di ciclo

   repeat *S1* **until** *cond*

   sia dato da:

   (costo del test *cond* + costo delle istruzioni in *S1*) * numero di volte in cui il ciclo è ripetuto;

Modello di costo temporale

5. il costo di un'istruzione di ciclo

   for $i \leftarrow 1$ **to** n **do** $S1$

   sia pari a:

   costo unitario per l'inizializzazione di i + (costo unitario del test $[i \leq n]$ di fine ciclo + costo delle istruzioni in $S1$ + costo di incremento di i) * n + costo unitario del test $[i \leq n]$ di fine ciclo;
6. il costo dell'accesso al valore di una variabile o a un elemento di un array sia nullo.

Modello di costo temporale

7. il costo di una chiamata di un sottoprogramma sia pari alla somma dei costi di esecuzione di tutte le istruzioni che lo compongono. In questo modo assumiamo che il costo di attivazione della procedura (passaggio di parametri, allocazione di variabili, ecc.) sia zero.

Le ipotesi precedenti eliminano la dipendenza dalla macchina e dal compilatore nell'analisi della complessità di un programma, ma portano a risultati approssimati.

Esempio

Consideriamo il seguente algoritmo che calcola il fattoriale:

fatt ← 1

for i ← 1 to n **do** fatt ← fatt*i;

Allora se *n*=3, il costo per il calcolo di 3! in base a questo algoritmo è:

inizializzazione di *fatt*	1 +
inizializzazione di *i* a 1	1 +
test $i \leq n$, quando $i \leq n$	(1 +
moltiplicazione	1 +
assegnazione a *fatt*	1 +
costo di incremento di i,	2) *
numero di volte in cui il ciclo è ripetuto	3 +
test $i \leq n$, quando $i > n$	1 =
	18

Operazioni dominanti

In generale, nel decidere come caratterizzare la complessità di tempo di un algoritmo, si considerano particolari operazioni che costituiscono il meccanismo *dominante* del procedimento risolutivo.

Esempi:

I confronti, gli scambi e gli spostamenti caratterizzano gli algoritmi di ordinamento.

Le operazioni di moltiplicazione caratterizzano il calcolo del fattoriale.

Le operazioni di addizione caratterizzano il calcolo di una sommatoria.

Operazioni dominanti

In generale, l'operazione dominante è caratterizzata dal fatto di essere "l'operazione eseguita più volte" nella computazione definita da un algoritmo.

Per individuare l'operazione dominante, sarà spesso sufficiente andare ad esaminare direttamente le operazioni contenute nei cicli più interni.

Esempio: Nel calcolo del fattoriale l'operazione dominante è la moltiplicazione.

Modello di costo temporale

Il modello di costo introdotto per un semplice linguaggio di programmazione imperativo ad alto livello sottintende un criterio di costo uniforme per le operazioni. Questo vuol dire che fra una operazione di assegnazione, di addizione, di moltiplicazione non si fa alcuna differenza.

Inoltre si assume che una cella di memoria possa contenere interi arbitrariamente grandi.

Non sempre simili semplificazioni sono accettabili.

Criterio di costo logaritmico

Per tenere conto del fatto che, in un calcolatore reale, una cella di memoria ha una capacità limitata, si assume che per contenere numeri arbitrariamente grandi occorre un numero di celle proporzionale al logaritmo dei numeri stessi e per eseguire una istruzione si deve pagare un costo proporzionale al logaritmo degli operandi.

Così, l'incremento di una unità per 2^{10} ha costo 10, mentre l'analogo incremento per 1 ha costo 1.

Criterio di costo logaritmico

Avevamo osservato che il seguente programma RAM che calcola x^y ha complessità in tempo O(y) sotto assunzione di costo uniforme, mentre sotto assunzione di costo logaritmico è O(y^2log$x$)

	READ	1	$l(1)+l(x)$	$O(\log x)$	1
	READ	2	$l(2)+l(y)$	$O(\log y)$	1
	LOAD	♯1	$l(1)$	$O(1)$	1
	STORE	3	$l(3)+l(1)$	$O(1)$	1
5	LOAD	2	$l(2)+l(y)$	$O(\log y)$	$y+1$
	JZERO	14	$l(y)$	$O(\log y)$	$y+1$
	LOAD	1	$l(1)+l(x)$	$O(\log x)$	y
	MULT	3	$l(3)+l(x)+l(x^y)$	$O(\log x+y\log x)$	y
	STORE	3	$l(3)+l(x^y)$	$O(y\log x)$	y
	LOAD	2	$l(2)+l(y)$	$O(\log y)$	y
	SUB	♯1	$l(1)+l(y)$	$O(\log y)$	y
	STORE	2	$l(2)+l(y)$	$O(\log y)$	y
	JUMP	5	1	$O(1)$	y
14	WRITE	3	$l(3)+l(x^y)$	$O(y\log x)$	1
	HALT	1	$O(1)$	1	

In totale, $O(y^2\log x)$ è il costo del programma.

Criterio di costo logaritmico

La complessità in spazio è determinata degli interi memorizzati nei registri da 0 a 3. Sotto assunzione di costo uniforme la complessità in spazio è O(1). Sotto assunzione di costo logaritmico è $O(y\log x)$, dal momento che il più grande intero memorizzato in uno di questi registri è x^y e $l(x^y)=y\log x$.

Se considerassimo questo programma come eseguito da un reale processore, il costo uniforme sarebbe realistico solo se una singola parola del calcolatore potesse memorizzare un intero grande quanto x^y.

Criterio di costo logaritmico

D'altronde, se x^y fosse maggiore del massimo intero rappresentabile in una parola, anche la complessità in tempo logaritmica sarebbe piuttosto irrealistica, dal momento che si assume che due interi i e j possano essere moltiplicati in un tempo $O(l(i)+l(j))$, mentre ad oggi non è noto alcun metodo che consenta di eseguire la moltiplicazione di due interi di n bit con un costo computazionale $o(n\log n)$.

Riepilogando, se è ragionevole assumere che ciascun numero incontrato in una computazione possa essere memorizzato in una parola, allora l'assunzione di costo uniforme è appropriata. Altrimenti una analisi più realistica dovrà basarsi su un modello di costo logaritmico.

Criterio di costo logaritmico

Altro esempio:

Il costo del calcolo di $n!$ è n sotto assunzione di costo uniforme (si fanno n moltiplicazioni).

Il costo approssimato è $\sum_{i=1}^{n} i \log i$

Approssimazione di Stirling: $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

Modello di costo in spazio

Detto A un array di 10 elementi di tipo intero, allora l'algoritmo che calcola la sommatoria dei numeri in A:

somma ← 0
for i ← 1 to n **do** somma ← somma + A[i];

ha la seguente complessità in spazio:

somma 1 +
i 1 +
A 10 =
12

Modello di costo in spazio

Come per la complessità in tempo, anche in questo caso approssimeremo la complessità in spazio considerando solo i dati strutturati e trascurando i dati di tipo semplice.

Inoltre, anche in questo caso si sta facendo una assunzione di costo uniforme.

Complessità in tempo

L'interesse maggiore nell'ambito della teoria della complessità è rivolto alla complessità in tempo, e ciò per due motivi:

1. la tecnologia attuale mette a disposizione memorie di grande ampiezza a costi contenuti;
2. la memoria, a differenza del tempo, è una risorsa che può essere riutilizzata.

Definire la complessità: in funzione di cosa?

Oltre alla definizione di un modello di costo, è necessario affrontare un altro problema per giungere ad un'analisi rigorosa della complessità di un algoritmo: individuare i *parametri* in funzione dei quali dev'essere definita la complessità.

A questo proposito si deve osservare che il tempo di computazione di un algoritmo dipende quasi sempre dai *dati di ingresso*.

<u>Esempio</u>: La complessità in tempo per il calcolo di 4! è 23, mentre è 18 il tempo necessario per calcolare 3! con lo stesso algoritmo.

Dimensione dell'input

In generale la complessità di un algoritmo è funzione della *dimensione dell'input*, ovvero della mole dei dati del problema da risolvere.

Si può adottare un criterio che prescinda dal tipo dei dati in ingresso (numeri, stringhe, vettori, ecc.), considerando come parametro la quantità di memoria utilizzata per rappresentare i dati in ingresso.

La rappresentazione dovrà essere "ragionevole" e non innaturalmente dilatata.

Dimensione dell'input

Nel caso di input numerico la scelta fra

1) valore dell'input

2) lunghezza della sua rappresentazione

può far apparire lineare il costo di un algoritmo che è invece esponenziale.

Esempio: calcolo del fattoriale di *n*. Le moltiplicazioni sono n, ma se la dimensione dell'input è log *n* allora il numero di moltiplicazioni è esponenziale in *n*.

Dimensione dell'input

Esempio: l'algoritmo che decide se x è un numero primo provando a dividere x per tutti i numeri naturali $i<x$ esegue $O(x)$ passi (divisioni). Assumendo, per semplicità, che una divisione sia eseguita in tempo costante rispetto al valore dei rispettivi operandi, la complessità risulterà esponenziale per una codifica binaria di x con $\lceil \log_2(x+1) \rceil$ bit, mentre sarà polinomiale (lineare, addirittura) se si utilizza una rappresentazione unaria (e.g., con x barre "|").

Si osservi che le le due codifiche non sono polinomialmente correlate.

Dimensione dell'input

Date due codifiche di un intero *x*, e(x) e e'(x), esse si diranno polinomialmente correlate se esistono due polinomi p, p' tali che, per ogni x

- $|e(x)| \leq p'(| e'(x) |)$
- $|e'(x)| \leq p(| e(x) |)$

dove |•| indica la lunghezza della codifica.

In seguito supporremo che la codifica dei dati in ingresso sia "ragionevole", ovvero:

- Considereremo codifiche in base $k \geq 2$ dei valori numerici
- Rappresenteremo insiemi mediante enumerazione delle codifiche dei loro elementi.
- Rappresenteremo relazioni e funzioni mediante enumerazione delle codifiche dei relativi elementi (coppie e n-ple)
- Rappresenteremo grafi come coppie di insiemi (di nodi e archi)

Dimensione dell'input

Anziché considerare la dimensione complessiva della rappresentazione dei dati si può assumere per semplicità un parametro caratterizzante la lunghezza dell'input.

<u>Esempio</u>: calcolo del valore di un polinomio in un punto. La dimensione dell'input è il grado del polinomio.

Funzione di complessità

Determinare la complessità in tempo (o in spazio) di un algoritmo significa determinare una *funzione di complessità f(n)* che fornisca la misura del tempo (o dello spazio di memoria occupato), al variare della dimensione dell'input, *n*.

Esempio:

Adottando come modello di costo quello basato su assunzione di costo uniforme, si evince che la funzione di complessità per l'algoritmo di calcolo del fattoriale è:

$$f(n) = 5n+3$$

Ancora sull'operazione dominante

Data la definizione di funzione di complessità *f(n)* si può chiarire ulteriormente il concetto di operazione dominante.

Diciamo che un'operazione in un algoritmo è dominante quando, detto $\tau(n)$ il numero di volte che essa viene ripetuta durante la realizzazione di un algoritmo in funzione della dimensione dell'input *n*, verifica

$$c \cdot \tau(n) \geq f(n)$$

dove *c* è una opportuna costante.

Termini dominanti

La teoria della complessità si occupa in particolare di studiare l'andamento asintotico delle funzioni di complessità $f(n)$.

Il vantaggio è che lo studio asintotico fornisce un criterio chiaro per poter confrontare due algoritmi.

Esempio:

A1: $f_1(n) = 3n^2 - 4n+2$

A2: $f_2(n) = 2n+3$

Per n=1,2 risulta $f_1(n) < f_2(n)$, mentre per n=3,4, ... risulta $f_1(n) > f_2(n)$. Come stabilire, dunque, quale sia l'algoritmo più efficiente? L'analisi asintotica ci consente di rispondere al quesito: A2 è più efficiente perché $\forall n>2 : f_1(n) > f_2(n)$.

Termini dominanti

Nello studio asintotico vengono trascurate le costanti moltiplicative e, se la $f(n)$ risulta costituita da più termini, vengono trascurati quei termini la cui crescita asintotica sia inferiore a quella di altri termini.

Esempio

Le seguenti funzioni di complessità:

$f_1(n) = 3n^2 + n+1$

$f_2(n) = 4n^2$

$f_3(n) = 4n^2 + 2n$

sono asintoticamente equivalenti, in quanto crescono tutte con il quadrato di n.

Nello studio della complessità asintotica degli algoritmi si ricorre alle seguenti definizioni di *ordine di grandezza*.

Ordine O

Def. di ordine O (o grande)

Una funzione $f(n)$ è di ordine $O(g(n))$ se esistono due costanti positive c ed n_0 tali che:

$$\forall n \geq n_0 \ f(n) \leq cg(n).$$

La $g(n)$ indica un limite *superiore* al comportamento asintotico della funzione f, ovvero f non si comporta asintoticamente peggio di g.

Esempio: La seguente funzione di complessità:

$$f(n) = 3n^2 + 3n - 1$$

è un $O(n^2)$ (c=4, n_0=3) In generale, $f(n)$ è un $O(n^k)$ per ogni $k \geq 2$. Fra le varie funzioni $g(n)$ tali che il costo dell'algoritmo sia un $O(g(n))$ si è generalmente interessati alla minorante.

$O(g(n))$ denota l'insieme di tutte le funzioni $f(n)$ di ordine $O(g(n))$.

Ordine di un polinomio

Proposizione. Se $f(n)= a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0$ è un polinomio di grado m, allora $f(n)$ è un $O(n^m)$.

Dimostrazione.

$$f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0 =$$

$$= \left(a_m + \frac{a_{m-1}}{n} + \cdots + \frac{a_0}{n^m} \right) n^m \leq$$

$$\leq \left(|a_m| + \frac{|a_{m-1}|}{n} + \cdots + \frac{|a_0|}{n^m} \right) n^m \leq$$

$$\leq (|a_m| + |a_{m-1}| + \ldots + |a_1| + |a_0|) n^m$$

Quindi posto $c=|a_m| + |a_{m-1}| + \ldots + |a_1| + |a_0|$ ed n_0=1 si ha:

$\forall n \geq 1 \ f(n) \leq cn^m$.

Legge transitiva per O

La relazione “è O grande di” soddisfa la proprietà transitiva, cioè se $f(n)$ è $O(g(n))$ e $g(n)$ è $O(h(n))$ abbiamo anche che $f(n)$ è $O(h(n))$.

$f(n)$ è $O(g(n))$ ➔ $\forall n \geq n_1 \ f(n) \leq c_1 g(n)$.

$g(n)$ è $O(h(n))$ ➔ $\forall n \geq n_2 \ g(n) \leq c_2 h(n)$.

Basta prendere come $n_0 = \max(n_1, n_2)$ e $c = c_1 \cdot c_2$.

Funzioni incommensurabili

Non sempre due funzioni $f(n)$ e $g(n)$ possono essere confrontate mediante un O-grande, cioè non sempre si può affermare che $f(n)$ è $O(g(n))$ oppure $g(n)$ è $O(f(n))$. Ci sono coppie di funzioni incommensurabili, nessuna delle quali è O-grande dell’altra. Quindi O-grande definisce una relazione d’ordine *parziale*.

Esempio:

$$f(n) = \begin{cases} n & n \text{ dispari} \\ n^2 & n \text{ pari} \end{cases} \qquad g(n) = \begin{cases} n & n \text{ pari} \\ n^2 & n \text{ dispari} \end{cases}$$

Ordine o

Def. di ordine o (o piccolo)

Una funzione *f(n)* è di ordine o(*g(n)*) se per ogni costante positiva *c* esiste un n_0 tale che:

$$\forall n \geq n_0 \ f(n) < cg(n).$$

La *g(n)* è un limite *superiore stretto*, per cui *f(n)=2n* è un O(*n*) ma non un o(*n*). Intuitivamente, nella notazione-o, la funzione *f(n)* diventa insignificante rispetto a *g(n)* man mano che *n* cresce; cioè:

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

Ordine Ω

Def. di ordine Ω (omega grande)

Una funzione *f(n)* è di ordine Ω(*g(n)*) se esistono due costanti positive *c* ed n_0 tali che:

$$\forall n \geq n_0 \ \ cg(n) \leq f(n).$$

La *g(n)* indica un limite *inferiore* al comportamento asintotico della funzione *f*, ovvero *f* non si comporta asintoticamente meglio di *g*.

Esempio: La funzione di complessità:

$$f(n) = 3n^2 + 3n - 1$$

è un $\Omega(n^2)$ (c=1, n_0=1). In generale, *f(n)* è un (n^k) per ogni $k \leq 2$. Siamo tuttavia interessati alla maggiorante.

Ω(*g(n)*) denota l'insieme di tutte le funzioni *f(n)* di ordine Ω(*g(n)*).

Ordine ω

Def. di ordine ω (omega piccolo)

Una funzione *f(n)* è di ordine ω(*g(n)*) se per ogni costante positiva *c* esiste un n_0 tale che:

$$\forall n \geq n_0 \;\; cg(n) < f(n).$$

La *g(n)* indica un limite *inferiore stretto* al comportamento asintotico della funzione *f*, ovvero *f*, per cui possiamo dire che $f(n) = 3n^2$ è un ω(*n*) ma non un ω(n^2).

Quindi *f(n)* diventa arbitrariamente grande rispetto a *g(n)*, ovvero:

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

Ordine Θ

Def. di ordine Θ (theta)

Una funzione *f(n)* è di ordine Θ(*g(n)*) se essa è allo stesso tempo di ordine O(*g(n)*) e Ω(*g(n)*).

Quindi devono esistere tre costanti positive c_1, c_2 ed n_0 tali che:

$$\forall n \geq n_0 \;\; c_1 g(n) \leq f(n) \leq c_2 g(n).$$

Se *f(n)* è di ordine Θ(*g(n)*), allora *f* si comporta asintoticamente esattamente come la *g*, per cui l'andamento di *g* caratterizza precisamente quello di *f*. Formalmente possiamo anche scrivere:

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = c \qquad \text{con } c>0$$

Θ(*g(n)*) denota l'insieme di tutte le funzioni *f(n)* di ordine Θ(*g(n)*).

Risultati noti per somme di interi

Non sempre si riescono a scrivere direttamente delle funzioni di complessità in forma chiusa, ovvero come espressione analitica di *n*.

In diversi casi si giunge a formulazioni di *f*(n) in termini di somme di potenze di interi per le quali non è immediato stabilire l'ordine di complessità

Si riportano alcuni risultati noti relativi a somme di potenze di interi che ricorrono spesso nelle analisi di complessità degli algoritmi:

Risultati noti per somme di interi

a) $f(n) = \sum_{i=1}^{n} 1 = n$ $\quad$ *f(n)* è $\Theta(n)$

b) $f(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ $\quad$ *f(n)* è $\Theta(n^2)$

c) $f(n) = \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$ $\quad$ *f(n)* è $\Theta(n^3)$

In generale si ha che:

$$\sum_{i=1}^{n} i^k = \frac{n^{k+1}}{k+1} + \frac{n^k}{2} + \text{termini di ordine inferiore}$$

Quindi possiamo concludere che:

$f(n) = \sum_{i=1}^{n} i^k$ è di ordine $\Theta(n^{k+1})$.

Analisi di algoritmi ricorsivi

Si fa uso di *relazioni di ricorrenza* nel caso di analisi della complessità di algoritmi ricorsivi, cioè di algoritmi che richiamano se stessi su sotto insiemi di dati e che forniscono le soluzioni finali combinando le soluzioni parziali ottenute dalle loro esecuzioni sui sotto insiemi.

Esempio: consideriamo la seguente funzione che calcola ricorsivamente il fattoriale di un numero *n*:

```
function fatt(n)
    if n = 0 then
            fatt ← 1
        else
            begin
                m ← n - 1
                fatt ← fatt(m)*n
            end
```

Analisi di algoritmi ricorsivi

Se si indica con $f(n)$ la complessità in tempo della funzione *fatt(n)*, allora sotto assunzione di costo uniforme si ha che:

$f(n) = 2$ se $n = 0$

$f(n) = 1 +$ {confronto n = 0}

$f(n - 1) +$ {calcolo di fatt(n - 1)}

$2 +$ {calcolo di m}

2 {calcolo di fatt} se $n > 0$

Quindi la complessità della funzione è espressa mediante la seguente *relazione di ricorrenza*:

$f(n) = 2$ se $n = 0$

$f(n) = 5 + f(n - 1)$ se $n > 0$

Risolvere relazioni di ricorrenza

Le relazioni di ricorrenza possono essere viste come equazioni dove l'incognita è una funzione. In pratica vogliamo trovare una definizione di *f* in forma esplicita, data una in forma implicita.

Come fare?

Il metodo più semplice: espandere le relazioni stesse, sostituendo successivamente alla relazione di partenza per *f*(n) quella riscritta per *f*(n-1) finché non resta che una formula chiusa per *f*(n) comprendente *n* e costanti.

Esempio: nel caso dell'algoritmo del fattoriale si ha:

$f(n) = 5 + f(n-1) =$

$= 5 + (5 + f(n-2)) =$

...

$= 5 + (5 + (5 + ... + (5 + f(0)))) =$

$= 2 + 5*n$

Per relazioni di ricorrenza più complesse, questo metodo può essere inefficace.

Risolvere relazioni di ricorrenza

In generale, in un algoritmo espresso ricorsivamente, la complessità può essere formulata attraverso una relazione di ricorrenza in cui è riportato:

1) il costo di calcolo dei risultati parziali ottenuti mediante chiamate ricorsive
2) il costo del *lavoro di combinazione* dei risultati parziali svolto prima e dopo le chiamate ricorsive.

Possiamo definire vere e proprie classi di relazioni ricorrenza in base alle caratteristiche del lavoro di combinazione svolto. Di seguito forniamo direttamente l'ordine di complessità per alcune di queste classi di ricorrenza.

Lavoro di combinazione indipendente da *n*

a) relazioni lineari di ordine *h*

$f(1)=c_1 \ f(2)=c_2 \ ... \ f(h)=c_h$

$f(n)= a_1 f(n-1) + a_2 f(n-2) + ... + a_h f(n-h) + b$ per $n > h$

Allora $f(n)$ è $O(a^n)$ con $a > 1$, ovvero f è esponenziale.

b) relazioni lineari di ordine h, con $a_h = 1$ e $a_j = 0$ per $1 \leq j \leq h-1$

$f(1)=c_1 \ f(2)=c_2 \ ... \ f(h)=c_h$

$f(n)= f(n-h) + b$ per $n > h$

Allora $f(n)$ è $O(n)$, cioè è lineare.

c) relazioni con partizione dei dati

$f(1)=c$

$f(n)= a \cdot f(n/k) + b$ per $n=k^m$, m intero e $n>1$

Se $a = 1$ allora $f(n)$ è $O(\log n)$.

Se $a > 1$ allora $f(n)$ è $O(n^{\log_k n})$.

Lavoro di combinazione proporzionale a *n*

a) relazioni lineari di ordine h, con $a_h = 1$ e $a_j = 0$ per $1 \leq j \leq h-1$

$f(1)=c_1 \ f(2)=c_2 \ ... \ f(h)=c_h$

$f(n)= f(n-h) + bn + d$ per $n > h$

Allora $f(n)$ è $O(n^2)$.

b) relazioni con partizione dei dati

$f(1)=c$

$f(n)= a \cdot f(n/k) + bn + d$ per $n=k^m$, m intero e $n>1$

Se $a < k$ allora $f(n)$ è $O(n)$.

Se $a = k$ allora $f(n)$ è $O(n \log n)$.

Se $a > k$ allora $f(n)$ è $O(n^{\log_k a})$.

Divide et Impera

- Una tecnica di risoluzione dei problemi consiste nella scomposizione di un problema in sottoproblemi, alcuni dei quali dello stesso tipo del problema di partenza, ma più semplici, e nella successiva combinazione delle soluzioni dei sottoproblemi in modo da ottenere la soluzione del problema di partenza.
- Questa tecnica, spesso usata ricorsivamente, consente di trovare algoritmi efficienti. Lo scheletro di una procedura ricorsiva DIVIDE-ET-IMPERA per risolvere un problema *P* di dimensione *n* è il seguente:

Divide et Impera

```
procedure DIVIDE-ET-IMPERA(P,n);
begin
   if n<=k then
        {risolvi P direttamente}
   else
        begin
          {dividi P in h sottoproblemi P_1, …, P_h di dimensione
           n_1, …, n_h, con n_i<n}
          for i := 1 to h do DIVIDE-ET-IMPERA(P_i,n_i)
          {combina i risultati di P_1, …, P_h in modo da ottenere
           la soluzione per P}
        end
end
```

Divide et Impera

Il calcolo della complessità di algoritmi progettati secondo la tecnica "divide et impera" è effettuato mediante la relazione di ricorrenza:

$$\begin{cases} f(n) = \text{costante} & \text{per } n \leq k \\ f(n) = d(n) + c(n) + \sum_{i=1}^{h} f(n_i) & \text{per } n > k \end{cases}$$

dove *d(n)* e *c(n)* denotano la complessità rispettivamente della divisione del problema e della composizione dei suoi risultati.

Se i dati sono partizionati in maniera bilanciata, cioè tutti gli n_i sono all'incirca uguali, allora l'algoritmo può risultare molto efficiente.

Esempi di algoritmi ottenuti mediante questa tecnica sono la *ricerca binaria*, il *quicksort* e il *mergesort*.

Complessità nel caso pessimo, ottimo e medio

Molte volte la complessità di un algoritmo non può essere caratterizzata da una sola funzione di complessità. A parità di dimensione dei dati, il tempo di calcolo può dipendere dalla specifica *configurazione* dei dati.

Esempio: Il seguente algoritmo determina la posizione del primo valore *false* in un array le cui componenti sono di tipo booleano:

```
i ← 1
trovato ← A[1]
while trovato do
    begin
    i ← i + 1
    if i > n then trovato ← false  else trovato ← A[i]
    end
result ← i
```

La dimensione dei dati è data dal numero di elementi di A, cioè *n*.

Complessità nel caso pessimo, ottimo e medio

L'operazione dominante rispetto alla quale valutare la complessità è il confronto.

La complessità in tempo non dipende solo da *n*.

Nel caso in cui i dati sono distribuiti in modo tale che A[1]=false, si ha:

$$f(n) = 1$$

Nel caso in cui l'array A non contiene un valore *false* si ha:

$$f(n) = 1 + 2n$$

Complessità nel caso pessimo, ottimo e medio

Di solito si considerano tre tipi di complessità:

- caso *pessimo*: si ottiene considerando quella particolare configurazione che, a parità di dimensione dei dati, dà luogo al massimo tempo di calcolo;
- caso *ottimo*: si ottiene considerando la configurazione che dà luogo al minimo tempo di calcolo;
- caso *medio*: si riferisce al tempo di calcolo mediato su tutte le possibili configurazioni dei dati, sempre per una determinata dimensione dei dati.

Nell'esempio precedente, si ha:

$$f_{ott}(n) = 1$$

$$f_{pess}(n) = 1 + 2n$$

Complessità nel caso pessimo, ottimo e medio

La determinazione della funzione di complessità nel caso medio è un po' più complicata, e richiede la conoscenza della *distribuzione di probabilità* sulle possibili configurazioni dei dati.

Esempio: Ricerca del primo false. Supponendo che tutte le configurazioni di *false* e *true* siano ugualmente probabili (assunzione di distribuzione uniforme), si ha 1/2 di probabilità che false si trovi al primo posto.

Complessità nel caso pessimo, ottimo e medio

1	*false*	*false*	*false*	*false*
2	*false*	*false*	*false*	*true*
3	*false*	*false*	*true*	*false*
...				
8	*false*	*true*	*true*	*true*
9	*true*	*false*	*false*	*false*
10	*true*	*false*	*false*	*true*
...				
16	*true*	*true*	*true*	*true*

Complessità nel caso pessimo, ottimo e medio

Indicato con E_1 l'evento:

$$E_1: A[1] = false$$

allora si avrà:

$$P(E_1) = 1/2$$

Nel caso si verifichi E_1 il costo computazionale è dato da:

$$costo(E_1) = 1$$

Analogamente, si ha 1/4 di probabilità che *false* non si trovi al primo posto ma al secondo. Indicato con E_2 l'evento:

$$E_2: A[1] = true \wedge A[2] = false$$

allora si avrà:

$$P(E_2) = 1/4$$

e si può facilmente mostrare che il costo computazionale è dato da:

$$costo(E_2) = 3$$

Complessità nel caso pessimo, ottimo e medio

In generale, l'evento:

$$E_k: A[1] = true \wedge A[2] = true \wedge \ldots \wedge A[k-1] = true \wedge A[k] = false$$

con $k \leq n$, ha probabilità $1/2^k$ di verificarsi:

$$P(E_k) = 1/2^k$$

e inoltre:

$$costo(E_k) = 1 + 2(k - 1)$$

Infine la probabilità di avere tutti true in A è $1/2^n$, dal momento che esiste una sola configurazione sulle 2^n possibili in cui ciò si verifica. Quindi, l'evento

$$E_{n+1}: A[1] = true \wedge \ldots \wedge A[n] = true$$

ha probabilità e costo seguenti:

$$P(E_{n+1}) = 1/2^n$$

$$costo(E_{n+1}) = 1 + 2n$$

Complessità nel caso pessimo, ottimo e medio

Allora il costo medio (o *costo atteso*) della ricerca del primo *false* in A è dato dalla *media pesata* dei costi dei singoli eventi E_i, dove i pesi sono dati dalle probabilità degli eventi stessi:

$$f_{med}(n) = \sum_{i=1}^{n+1} P(E_i) \cdot \text{costo}(E_i) =$$

$$= \left(1 + \frac{1}{2} + \frac{1}{4} \cdot + \cdots + \frac{1}{2^k} + \cdots + \frac{1}{2^n}\right) - 1 + \frac{1}{2^n} + \left(\frac{1}{2} + \cdots + \frac{k-1}{2^k} + \cdots + \frac{n-1}{2^{n-1}} + \frac{n}{2^n}\right) + \frac{n}{2^n} =$$

$$= \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 3 + \cdots + \frac{1}{2^k}(1 + 2(k-1)) + \cdots + \frac{1}{2^n}(1 + 2(n-1)) + \frac{1}{2^n}(1 + 2n) =$$

$$= \frac{1}{2} \cdot (1+0) + \frac{1}{4} \cdot (1+2) + \cdots + \frac{1}{2^k}(1 + 2(k-1)) + \cdots + \frac{1}{2^n}(1 + 2(n-1)) + \frac{1}{2^n}(1 + 2n) =$$

$$= \frac{1}{2} + \frac{1}{4} \cdot + \cdots + \frac{1}{2^k} + \cdots + \frac{1}{2^n} + \frac{1}{2^n} + \frac{2}{4} + \cdots + \frac{2(k-1)}{2^k} + \cdots + \frac{2(n-1)}{2^n} + \frac{2n}{2^n} =$$

$$= \left(1 + \frac{1}{2} + \frac{1}{4} \cdot + \cdots + \frac{1}{2^k} + \cdots + \frac{1}{2^n}\right) - 1 + \frac{1}{2^n} + \left(\frac{1}{2} + \cdots + \frac{k-1}{2^k} + \cdots + \frac{n-1}{2^{n-1}} + \frac{n}{2^n}\right) + \frac{n}{2^n} =$$

Complessità nel caso pessimo, ottimo e medio

$$= \left[\sum_{i=0}^{n} \left(\frac{1}{2}\right)^i\right] - 1 + \frac{1}{2^n} + \left(\sum_{i=1}^{n} \frac{i}{2^n}\right) + \frac{n}{2^n}$$

Si dimostra che: $\displaystyle\sum_{i=0}^{n} q^i = \frac{1-q^{n+1}}{1-q}$

da cui $\displaystyle\sum_{i=0}^{n} \left(\frac{1}{2}\right)^i = 2\left(1 - \frac{1}{2^{n+1}}\right)$

Inoltre si dimostra che: $\displaystyle\sum_{i=0}^{n} iq^i = \frac{nq^{n+2} - (n+1)q^{n+1} + q}{(1-q)^2}$

da cui $\displaystyle\sum_{i=0}^{n} i\left(\frac{1}{2}\right)^i = 2\left(\frac{n}{2^{n+1}} - \frac{n+1}{2^n} + 1\right)$

Complessità nel caso pessimo, ottimo e medio

quindi

$$f_{med}(n) = 2\left(1 - \frac{1}{2^{n+1}}\right) - 1 + \frac{1}{2^n} + 2\left(\frac{n}{2^{n+1}} - \frac{n+1}{2^n} + 1\right) + \frac{n}{2^n} = 3 - \frac{1}{2^n - 1}$$

Si osservi che $\forall n \;\; 2 \leq f_{med}(n) \leq 3$

e che $\lim_{n \to +\infty} f_{med}(n) = 3$

quindi $f_{med}(n)$ è di ordine $\Theta(c)$, dove c è una qualsiasi costante maggiore di 0. Riepilogando:

- caso pessimo: $\Theta(n)$;
- caso ottimo: $\Theta(2)$;
- caso medio: $\Theta(3)$.

Complessità nel caso pessimo, ottimo e medio

In alcuni algoritmi tutte le possibili configurazioni dei dati sono equivalenti per quanto riguarda la complessità in tempo. In questo caso la complessità ottima, pessima e media coincidono.

Esempio: Il seguente algoritmo:

counter ← 1

for i ← 1 **to** n **do**

if A[i] **then** counter ← counter + 1 ;

result ← counter

deve necessariamente analizzare tutti gli elementi di A per contare il numero di *true*. Quindi:

$$f_{ott}(n) = f_{pess}(n) = f_{med}(n)$$

Le classi di complessità computazionale

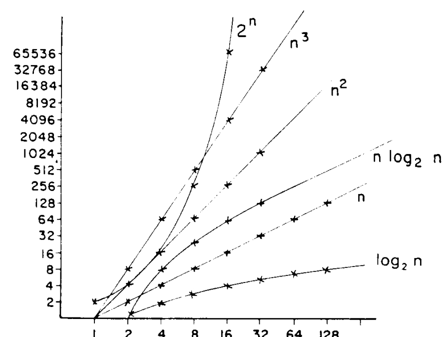
L'analisi della complessità asintotica degli algoritmi consente di categorizzare gli stessi in *classi di complessità*:

costante	$\Theta(1)$
logaritmica	$\Theta(\log n)$
lineare	$\Theta(n)$
nlog	$\Theta(n \log n)$
quadratica	$\Theta(n^2)$
cubica	$\Theta(n^3)$
esponenziale	$\Theta(a^n)$ con $a>1$

Gli algoritmi con complessità costante sono più efficienti di quelli con complessità logaritmica, che a loro volta sono più efficienti di quelli con complessità lineare, e così via. Per indicare ciò si scrive anche che:

$$\Theta(1) < \Theta(\log n) < \Theta(n) < \Theta(n \log n) < \Theta(n^2) < \Theta(n^3) < \Theta(a^n)$$

Le classi di complessità computazionale



Si osservi che la crescita di funzioni di complessità di ordine $\Theta(n)$ o $\Theta(n \log n)$ è molto più lenta che per funzioni di complessità di ordine superiore.

Le classi di complessità computazionale

Un algoritmo con complessità esponenziale sarà praticamente utilizzabile solo per valori piccolissimi di *n*, come mostra chiaramente la seguente tabella:

$log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
1	2	2	4	8	4
3,322	10	33,22	10^2	10^3	$> 10^3$
6,644	10^2	664,4	10^4	10^6	$>>10^{25}$
9,966	10^3	9.966,0	10^6	10^9	$>>10^{250}$
13,287	10^4	132.877	10^8	101^2	$>>10^{2500}$

Le classi di complessità computazionale

Per algoritmi di complessità esponenziale l'aumento della velocità di calcolo di un elaboratore non serve a molto, poiché ciò consente di risolvere nello stesso tempo problemi solo di poco più complessi di quelli risolvibili precedentemente.

complessità in tempo	calcolatori attuali	calcolatori 1000 volte più veloci
n	n_1	$1000\ n_1$
n^2	n_2	$31,6\ n_2$
n^3	n_3	$10\ n_3$
a^n	n_4	$n_4 + 9,97$

La tabella riporta i tempi di computazione relativi a diverse funzioni di complessità ed a diverse dimensioni dei dati di ingresso, nell'ipotesi che un'operazione elementare venga eseguita in un μsec. Un aumento di 1000 volte della velocità di calcolo farebbe crescere solo di 10 unità la dimensione delle istanze di un problema risolubili in un'ora con un algoritmo di complessità 2^n.

Le classi di complessità computazionale

Benché sia interessante studiare gli algoritmi di ogni ordine di complessità, la suddivisione che riveste la massima importanza è quella tra algoritmi *polinomiali* ed *esponenziali*.

Gli algoritmi polinomiali hanno complessità $O(n^k)$, con $k>0$, mentre gli algoritmi esponenziali hanno complessità $\Omega(a^{g(n)})$, con $a>1$ e $g(n)$ funzione crescente di n.

Quindi sono polinomiali gli algoritmi di complessità $\Theta(\log n)$ e $\Theta(n^4)$, mentre sono esponenziali gli algoritmi $\Theta(2^{\log n})$, $\Theta(2^n)$, e $\Theta(n^{\log n})$.

La suddivisione tra le due famiglie indica, in linea di principio, quali algoritmi abbiano importanza pratica e quali no.

Tesi della computazione sequenziale

Quesito: è possibile che algoritmi polinomiali per un modello di calcolo sequenziale risultino esponenziali per un altro modello e viceversa?

Tesi della computazione sequenziale: *calcolatori diversi possono simularsi a vicenda (secondo quanto afferma la tesi di Church) senza però che un algoritmo cambi classe (polinomiale o esponenziale) per effetto della simulazione.*

La tesi di Church afferma in pratica che tutti gli elaboratori sono in grado di risolvere gli stessi problemi

La tesi della computazione sequenziale è più forte, in quanto afferma che la complessità degli algoritmi non cambia significativamente da calcolatore a calcolatore (qui per cambio significativo intendiamo che si passa da complessità polinomiale ad esponenziale o viceversa).

Tesi della computazione sequenziale

Alla base della tesi della computazione sequenziale c'è l'ipotesi ragionevole che il rapporto tra i tempi di esecuzione di due operazioni elementari equivalenti, su due calcolatori diversi A e B, sia al più una funzione polinomiale $r(n)$. Ne segue che un algoritmo di complessità polinomiale $p(n)$ su A, avrà al più complessità polinomiale $p(n)r(n)$ su B.

Tesi della computazione sequenziale

Alla base della tesi della computazione sequenziale c'è l'ipotesi ragionevole che il rapporto tra i tempi di esecuzione di due operazioni elementari equivalenti, su due calcolatori diversi A e B, sia al più una funzione polinomiale $r(n)$. Ne segue che un algoritmo di complessità polinomiale $p(n)$ su A, avrà al più complessità polinomiale $p(n)r(n)$ su B.