i	*numero*	*sommapos*	*sommaneg*
1	3	0	3
2	15	0	18
3	-5	-5	18
4	3	-5	21
5	-9	-14	21
6	0	-14	21
7	12	-14	33

Fig. 16 - Stampe parziali per il programma di fig. 15

Se ora rieseguiamo il programma con gli stessi dati della prova precedente otteniamo la stampa di fig. 16.

I dati ottenuti mostrano che il secondo errore continua a presentarsi. Dalle nuove informazioni ottenute è ora chiara la causa dell'errore: l'inversione di ruolo tra le due variabili. Possiamo allora correggere il programma modificando opportunamente la espressione booleana nella istruzione **if**; si ottiene in tal modo il programma corretto (non riproduciamo il programma, peraltro facilmente ottenibile da quello di fig. 15).

Boffi et al : Fondamenti di programmaz.

6.
Analisi di programmi: la complessità

La valutazione dell'efficienza degli algoritmi e la ricerca di algoritmi sempre più efficienti sono gli obiettivi di quella parte dell'informatica nota come *complessità di calcolo;* essa costituisce l'argomento di questo capitolo.

Nei primi due paragrafi verrano esaminati i concetti fondamentali che costituiscono le basi per analizzare i programmi dal punto di vista delle risorse richieste.

Nei paragrafi successivi questi concetti verranno applicati a due diversi problemi. In particolare, nel par. 3 verrà considerato il problema della gestione di una tavola, e, nel par. 4, il problema dell'ordinamento di un insieme. Nel paragrafo finale faremo alcune considerazioni conclusive sul progetto di programmi efficienti.

1. EFFICIENZA DEI PROGRAMMI

Un programma è tanto più efficiente quanto minore è l'utilizzo di risorse di calcolo necessarie per la sua esecuzione. Le risorse di calcolo che si considerano sono due: il tempo di calcolo e la quantità di memoria necessari per l'elaborazione. Nel seguito considereremo prevalentemente la risorsa *tempo,* valutando così il costo o la *complessità* del programma. Pertanto diremo che un programma è tanto più efficiente quanto meno tempo viene richiesto per la sua esecuzione.

Un primo modo di valutare il costo di un programma è quello di esprimere il tempo di calcolo con unità di misura solari, come, ad esempio, i secondi. In questo caso la valutazione del costo di un programma è, in teoria, molto semplice: basta eseguire il programma disponendo di un orologio con cui

misurare il tempo impiegato. Inoltre, se si vogliono confrontare due programmi e stabilire quale sia il più efficiente, è sufficiente eseguirli e confrontarne i rispettivi tempi di esecuzione per individuare il migliore.

Anche se questo metodo di valutazione è apparentemente corretto, i risultati che si ottengono non sono validi perché dipendono in modo essenziale dalle particolari condizioni in cui si effettuano le prove. Infatti essi dipendono:

1. dalla macchina e dal compilatore utilizzati: il confronto di due programmi è valido solo se li traduciamo con il medesimo compilatore e li eseguiamo con lo stesso elaboratore;
2. dai dati di ingresso ai due programmi. Per trarre una conclusione valida è necessario che i due programmi ricevano in ingresso gli stessi dati. Inoltre per stabilire quale dei due programmi sia più efficiente non è sufficiente eseguire i due programmi una sola volta, ma più volte con dati differenti. In questo caso è necessario stabilire dopo quante esecuzioni possiamo terminare il confronto.

Le osservazioni precedenti portano alla conclusione che non è possibile valutare il costo di un programma in unità di tempo solari, perché in tal modo non si effettua un'analisi sufficientemente rigorosa e completa del suo costo. Nel seguito vedremo come sia possibile esprimere il costo di un programma tramite una funzione che non dipende dal particolare sistema di elaborazione a disposizione, dal compilatore usato e dai particolari dati di ingresso utilizzati. In questo modo è possibile ottenere valutazioni oggettive; tuttavia i risultati che questo metodo fornisce sono approssimati, perché esso si basa su alcune ipotesi semplificative, che verranno considerate in dettaglio nel seguito di questo paragrafo.

1.1. Il modello di costo

L'analisi della complessità di un programma è basata sulla ipotesi che il costo di esecuzione di ogni istruzione semplice e di ogni operazione di confronto sia pari a una unità di costo indipendentemente dal linguaggio e dal sistema usato.

Il costo delle altre istruzioni dipende dal numero di istruzioni semplici o di test che sono richiesti. In particolare, nel caso del linguaggio Pascal si assume che:

1. il costo di esecuzione di ogni istruzione semplice (istruzione di assegnazione, di lettura, di scrittura) è 1;

2. il costo di un'istruzione composta è pari alla somma dei costi delle istruzioni che la compongono;
3. il costo di un'istruzione di ciclo è dato dalla somma del costo totale di esecuzione del test di fine ciclo e dal costo totale di esecuzione del corpo del ciclo. Assumiamo che il costo di esecuzione del test sia unitario e, quindi, il primo termine è pari al numero di volte che il ciclo viene ripetuto. Il secondo termine è pari alla somma dei costi di esecuzione delle istruzioni del corpo del ciclo tenendo conto di quante volte ciascuna istruzione del ciclo è eseguita;
4. il costo di un'istruzione di tipo **if** ... **then** ...è dato dal costo di esecuzione del test (che assumiamo unitario) più il costo di esecuzione della istruzione che segue la parola **then**, se la condizione è vera. In modo analogo si valuta il costo di esecuzione di un'istruzione **if** ...**then** ... **else;**
5. il costo di una attivazione di una procedura (o di una funzione) è pari alla somma dei costi di esecuzione di *tutte* le istruzioni che la compongono tenendo eventualmente conto del fatto che all'interno della procedura (o funzione) possono essere presenti altre attivazioni di procedure o funzioni. In questo modo assumiamo che il costo di attivazione della procedura (passaggio di parametri, allocazione di variabili locali 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. Infatti, si assume che il costo dell'istruzione di assegnazione:

$$A := A + 1$$

in cui la valutazione dell'espressione richiede una addizione, sia pari al costo dell'istruzione

$$A := B * (C + 3) + ((D + 19) \textbf{ div } (5 + A * 2)) + 2 * E$$

in cui la valutazione dell'espressione richiede 5 addizioni, 3 moltiplicazioni e una divisione. Questo chiaramente non è vero: la seconda istruzione comporta l'esecuzione di un numero maggiore di operazioni; si noti inoltre che le operazioni di moltiplicazione e di divisione richiedono un maggior tempo di calcolo della addizione.

Osserviamo però che, se indichiamo con $t(1)$ il tempo di esecuzione della prima istruzione e con $t(2)$ quello della seconda istruzione, allora esiste una costante intera c tale che

$$c\, t(1) > t(2)$$

In altre parole $t(2)$ è maggiore di $t(1)$ al più per un fattore costante c, e, quindi, l'ipotesi di assumere il costo di ogni istruzione pari a 1 è esatta a meno di un fattore costante. Un ragionamento analogo può essere fatto per ogni istruzione di un qualunque programma e, quindi, possiamo concludere che *il costo di un programma viene valutato a meno di un fattore costante.*

Un altro vantaggio di queste semplificazioni è quello di poter prescindere dal linguaggio di programmazione in cui viene scritto il programma e di poter far riferimento direttamente all'algoritmo descritto in linguaggio naturale. Nel seguito, parleremo del costo di esecuzione (o della complessità) di un algoritmo intendendo il costo di esecuzione di un programma, scritto in un qualunque linguaggio di programmazione, che lo realizza.

Il secondo problema che è necessario affrontare per giungere ad un'analisi rigorosa della complessità di un programma è stabilire in funzione di quali parametri deve essere definita la sua complessità.

Il costo di esecuzione di un programma dipende quasi sempre dai dati di ingresso; ad esempio, il tempo di esecuzione di un programma P di ordinamento di un insieme di numeri dipende dalle dimensioni dell'insieme che si considera: generalmente un programma impiega meno tempo per ordinare un insieme di 10 elementi che per ordinare un insieme di 10000 elementi.

Pertanto, per ottenere una valutazione rigorosa del tempo di esecuzione è necessario individuare la funzione $f(n)$, che esprime il numero delle istruzioni semplici e di test eseguite in funzione del numero n dei dati del problema, cioè in funzione della *dimensione dell'input*.

Ad esempio, la complessità di un programma di ordinamento può essere espressa da una funzione del tipo

$$n\,(n+1)/2 + n + 5$$

In questo caso si intende che $n\,(n+1)/2 + n + 5$ è il numero di operazioni e confronti richiesti per ordinare insiemi di n elementi.

In molti casi il costo di esecuzione del programma dipende non solo dalle dimensioni dell'input, ma anche dai particolari valori dei dati stessi. In particolare, è possibile distinguere diversi casi: il caso migliore, il caso peggiore, il caso medio. Nel seguito si valuterà la complessità facendo riferimento generalmente al caso peggiore e talvolta al caso medio.

Nella valutazione di un algoritmo dal punto di vista del caso peggiore si fa riferimento a quei valori dei dati di ingresso per cui il costo di esecuzione è maggiore. Invece, nella valutazione di un algoritmo dal punto di vista del caso medio, si valuta la media aritmetica dei costi delle esecuzioni rispetto a tutti i

possibili dati di ingresso.

L'esempio seguente illustra i concetti visti.

Esempio. Consideriamo il programma per la ricerca esaustiva di un elemento in una tavola con n elementi, rappresentata tramite un array.

In questo caso possiamo assumere che la dimensione dell'input sia data dal numero di elementi dell'array.

Consideriamo ora il programma di ricerca esaustiva in una tavola dato in fig. 1.

Il costo di esecuzione del programma dipende dalla posizione del particolare elemento che si vuole individuare: se l'elemento cercato è il primo della tavola allora si effettua un solo confronto; se l'elemento cercato è il secondo della tavola allora si effettuano due confronti e così via. Il caso peggiore è costituito dalla ricerca dell'ultimo elemento o da una ricerca infruttuosa, perché in questo caso l'algoritmo esamina tutte le componenti dell'array ed esegue il ciclo n volte.

Valutiamo ora il costo del programma nel caso di ricerca dell'ultimo elemento:

- l'istruzione 1 è eseguita 1 volta;
- l'istruzione 3 è eseguita n volte;
- il test dell'istruzione **repeat** è eseguito n volte;
- il test dell'istruzione **if then else** è eseguito 1 volta;
- l'istruzione in 6 è eseguita 1 volta;
- l'istruzione in 7 non viene eseguita.

Pertanto il costo di esecuzione del programma è

$$1 + n + n + 1 + 1 = 3 + 2\,n$$

È facile vedere che la stessa funzione esprime anche il costo del programma nel caso di ricerca infruttuosa.

Se si vuole valutare il comportamento del programma nel caso medio, è necessario distinguere il caso di ricerca con successo da quello di ricerca infruttuosa. Nel caso di ricerca fruttuosa, se si assume che tutti gli elementi dell'array possano essere ricercati con uguale probabilità pari a $1/n$, allora è facile vedere che il programma richiede mediamente $(n+1)/2$ confronti. Infatti, se ricerchiamo l'i-esimo elemento si effettuano i confronti e, quindi, il numero di confronti medio è dato da

$$\Sigma_{1\le i\le n}\,\mathrm{Prob(E(i))}\;i = \Sigma_{1\le i\le n}\,i/n = (n+1)/2$$

dove Prob(E(i)) è la probabilità che l'elemento da cercare nell'array sia quello

```
procedure ricerca_esaustiva (var t : tipotavola;
                             k : tipochiave; var trovato : boolean);
{la procedura ricerca nella tavola l'elemento avente chiave k; se la ricerca
ha successo allora la variabile booleana trovato viene posta a true,
altrimenti a false}
var i : integer;
begin
{1}   i := 0;
{2}   repeat
{3}         i := i +1
{4}   until (t[i].chiave = k) or (i = n);
{5}   if (t[i].chiave = k)
{6}   then trovato := true
{7}   else trovato := false
end;
```

Fig. 1 - Programma per la ricerca esaustiva in una tavola

in posizione i.

Osserviamo inoltre che, nel caso di ricerca infruttuosa, il programma richiede ogni volta di effettuare n confronti.

1.2. Comportamento asintotico

Individuare esattamente la funzione che esprime il costo di un algoritmo è, nella maggioranza dei casi, molto difficile. Generalmente, per semplificare l'analisi, non si cerca di individuare esattamente la funzione che esprime il costo del programma ma si ritiene sufficiente stabilire il suo comportamento asintotico quando le dimensioni dell'input tendono all'infinito. Più precisamente, si cerca di valutare l'andamento della funzione che esprime il costo del programma a meno di costanti moltiplicative e di termini additivi di ordine inferiore. Diremo, ad esempio, che un programma o un algoritmo ha complessità lineare se il suo costo in funzione delle dimensioni dell'input è

$$c_1 n + c_2$$

dove c_1 e c_2 sono costanti intere (n rappresenta le dimensioni dell'input).

Ad esempio, due programmi i cui costi sono espressi rispettivamente dalle funzioni $(3 + n)$ e $(100\ n + 3027)$, sono caratterizzati dalla stessa complessità asintotica, e cioè da una complessità lineare. Si noti che la funzione $(n+3)$ esprime un costo minore della funzione $(100\ n + 3027)$ e, quindi, ignorare la costante moltiplicativa (pari in un caso a 1 e nell'altro a 100) e il termine additivo di ordine inferiore (pari in un caso a 3 e nell'altro a 3027) può essere in alcuni casi una semplificazione eccessiva. Pertanto, i risultati che si ottengono nella valutazione della complessità di un programma devono essere interpretati e valutati tenendo conto delle semplificazioni fatte. È importante però sottolineare che ignorando le costanti moltiplicative e i termini additivi di ordine inferiore, l'analisi per stabilire il costo di un algoritmo viene molto semplificata e le valutazioni ottenute, pur con questa approssimazione, permettono di giungere a risultati significativi nella maggioranza dei casi.

2. LA COMPLESSITÀ DI UN PROGRAMMA E DI UN PROBLEMA

Le osservazioni e le considerazioni del paragrafo precedente permettono di esprimere il costo di esecuzione di un programma, o di un algoritmo, come una funzione delle dimensioni dell'input in cui si ignorano le costanti moltiplicative. Nel resto del paragrafo verranno definite diversi tipi di delimitazioni che formalizzano il concetto di complessità di un programma (o di un algoritmo) e di un problema. Nel seguito si assume che n rappresenti il parametro in funzione del quale viene espresso il costo di un programma o di un algoritmo.

2.1. Le notazioni O e Ω

In base alle considerazioni svolte, la complessità di un programma è lineare quando il suo costo di esecuzione è, ad esempio, $2n$; analogamente, la complessità di un algoritmo è quadratica se il suo costo di esecuzione è una funzione quadratica nelle dimensioni dell'input, come ad esempio $n**2 + 2n$ (qui e nel seguito $n**i$, i intero, indica la potenza i-esima di n ; pertanto $n **2$ indica il quadrato di n e $2**n$ indica una funzione esponenziale).

In generale si utilizza la seguente definizione.

Definizione 1. Un programma (o un algoritmo) ha *costo* $O(f(n))$, o ha *complessità* $O(f(n))$, se esistono opportune costanti a, b e n' tali che il numero

di istruzioni $t(n)$ che vengono eseguite nel caso peggiore con input di dimensione n verifica

$$t(n) < a\,f(n) + b$$

per ogni $n > n'$.

Utilizzando la definizione precedente possiamo scrivere, ad esempio, che

$$50n **2 + 10n + 100 = O(n **2)$$
$$0.5\,n **3 + 1000\,n **2 = O(n **3)$$
$$2**n + 100 = O(2**n).$$

Se, invece, il programma richiede, per ogni possibile input, l'esecuzione di un numero costante di operazioni, allora la sua complessità è $O(1)$.

Si noti che nella definizione 1 è sufficiente ottenere una valutazione per eccesso. Quindi, in base alla definizione, se un programma ha costo $O(n)$ banalmente ha anche costo $O(n**2)$. È chiaro che in questo caso $O(n)$ esprime in modo più adeguato di $O(n**2)$ il costo del programma con complessità lineare. In generale tra le varie funzioni $f(n)$ tali che il costo del programma è $O(f(n))$ si cerca, quindi, la minorante.

La notazione O fornisce una delimitazione superiore al costo di esecuzione di un algoritmo, cioè fornisce una valutazione approssimata per eccesso.

Per specificare una delimitazione inferiore introduciamo la notazione Ω.

Definizione 2. Un programma (o un algoritmo) ha *costo* $\Omega(g(n))$, o ha *complessità* $\Omega(g(n))$ se esiste una opportuna costante positiva c tale che il numero di istruzioni $t(n)$ che vengono eseguite nel caso peggiore con input di dimensione n verifica

$$t(n) > c\,g(n)$$

per un numero infinito di valori di n.

Si noti la differenza tra le due definizioni di delimitazione superiore ed inferiore alla complessità di un algoritmo. In base alla prima definizione un algoritmo ha costo $O(f(n))$ se, *per ogni n e per ogni input di dimensione n,* impiega una quantità di risorse proporzionale a $f(n)$ (a meno di costanti additive). Invece, nella seconda definizione, un algoritmo ha costo $\Omega(g(n))$ se *esiste una sequenza infinita di istanze del problema aventi dimensioni crescenti,* ciascuna delle quali richiede una quantità di risorse maggiore di $c*g(n)$ (per un'opportuna costante c). In altre parole, sapere che un algoritmo ha costo $\Omega(g(n))$ non è sufficiente per affermare che, per ogni istanza del problema di dimensione n, il costo di esecuzione è almeno $c\,g(n)$.

La ragione di questa differenza è che capita frequentemente di avere algoritmi che si comportano efficientemente in molti casi ma non in tutti. La definizione 2 non considera i casi "facili", ma fa riferimento ai casi più costosi per stabilire la delimitazione inferiore del problema.

Chiaramente la delimitazione inferiore al costo di un algoritmo non può essere rappresentata da una funzione che, al crescere delle dimensioni dell'input, cresce più velocemente della delimitazione superiore della complessità. Abbiamo una valutazione esatta del costo di un algoritmo quando le due delimitazioni coincidono.

Nel seguito quando non è specificato espressamente per delimitazione della complessità si intende la delimitazione superiore.

Come abbiamo già osservato nel paragrafo precedente ignorare la costante moltiplicativa porta ad una valutazione approssimata perché permette di asserire che il costo di un programma è $O(n)$, cioè lineare, quando il numero delle istruzioni eseguite è, ad esempio, $2n$ o $1000n$. Tuttavia l'ipotesi semplificativa permette di valutare, almeno in modo approssimato, l'efficienza di un programma o di un algoritmo e di effettuare confronti di efficienza fra due programmi o algoritmi.

Per capire meglio l'ultima affermazione consideriamo la fig. 2, che esprime il costo di esecuzione di 4 programmi che risolvono lo stesso problema e aventi complessità $100\,n$, $10\,n**2$, $(n**3)/2$ e $2**n$, rispettivamente. Se supponiamo di utilizzare un elaboratore in grado di compiere un milione di operazioni al secondo, allora i diversi programmi risolvono in un microsecondo istanze del problema aventi dimensioni comparabili (circa 10). Al crescere delle dimensioni del problema questo non è più vero. Infatti i tempi necessari per eseguire istanze del problema aventi dimensioni 30 risultano molto diversi; la differenza risulta ancora più drammatica per dimensioni maggiori, come è illustrato dalla fig. 2.a. Ad esempio, il tempo richiesto per risolvere un'istanza di dimensione 90 con l'algoritmo esponenziale richiede un tempo di calcolo maggiore dell'età della terra.

La fig. 2.b evidenzia le dimensioni massime dei problemi che sono risolubili in un secondo, un minuto, un'ora o un giorno. Ad esempio, se si utilizza il programma avente costo $100n$, è possibile risolvere, in un minuto o in un'ora, istanze del problema aventi dimensione 600000 e 36000000, rispettivamente; con il programma di complessità $2**n$ in un minuto o in un'ora possiamo risolvere istanze aventi dimensioni 25 e 31, rispettivamente.

Complessità del programma	*Dimensioni dell'input*		
	10	30	60
100 n	1 millisec.	3 millisec.	6 millisec.
10 n **2	1 millisec.	9 millisec.	36 millisec.
n **3	1 millisec.	27 millisec.	2.1 secondi
2** n	1millisec.	18minuti	366 secoli

Fig. 2.a - La tabella indica il tempo di calcolo in funzione delle dimensioni del problema

Complessità del programma	*Tempo impiegato*			
	1 sec.	1 min.	1 ora	1 giorno
100 n	10^4	$6*10^5$	$36*10^6$	$864*10^6$
10 n **2	316	2449	18970	92951
n **3	100	392	1532	4420
2** n	20	25	31	36

Fig. 2.b - La tabella indica le dimensioni massime del problema risolubili in una data quantità di tempo

Fig. 2 - Confronto di diversi algoritmi con complessità diverse

2.2. Valutazione della complessità di un programma

La valutazione del costo di un programma o di un algoritmo richiede la determinazione di una delimitazione superiore ed una inferiore. In questo paragrafo presentiamo un insieme di regole che aiutano a trovare la delimitazione superiore della complessità.

Regola 1. Supponiamo che il programma sia composto di due parti P e Q da eseguire sequenzialmente e che i costi di P e Q siano $S(n)=O(f(n))$ e $T(n)=O(g(n))$. Allora il costo del programma è $O(\max(f(n),g(n)))$.

Per dimostrare la correttezza della regola osserviamo che in questo caso il costo complessivo del programma è, chiaramente, pari a

$$S(n) + T(n).$$

Inoltre si noti che

1. poiché $S(n) = O(f(n))$ allora esistono costanti a', b' e n' tali che $S(n) < a'f(n) + b'$ per $n>n'$;
2. poiché $T(n) = O(g(n))$ allora esistono costanti a'', b'' e n'' tali che $T(n) < a''g(n) + b''$ per $n>n''$.

Da queste due osservazioni deriva che, per $n > \max(n', n'')$

$$S(n) + T(n) < a'f(n) + b' + a'' g(n) + b''$$
$$<(a' + a'') \max(f(n),g(n)) + (b' + b'')$$

Quindi, $O(S(n) + T(n))$ è proprio $O(\max(f(n),g(n)))$.

La regola precedente può essere generalizzata nel seguente modo.

Regola 1'. Se un programma è composto da una successione finita di parti P(1), P(2), ... P(k) da eseguire sequenzialmente allora la complessità del programma è pari alla complessità del passo più costoso.

Sia dato, ad esempio, un programma composto da 3 parti P, Q, R che vengono eseguite sequenzialmente. Se P, Q e R hanno, rispettivamente, complessità $O(n**2)$, $O(n**3)$ e $O(n \log n)$, allora il costo complessivo del programma è pari a O(n**3).

Utilizzando la definizione 1 la regola 1 può essere così espressa:

Date due funzioni $f(n)$ e $g(n)$, se $f(n)=O(g(n))$, allora
$O(f(n) + g(n)) = O(g(n))$.

La seconda regola che presentiamo permette di valutare il costo del programma quando esso richiede più volte l'esecuzione di un insieme di istruzioni o l'attivazione di una procedura.

Regola 2. Supponiamo che un programma richieda per k volte l'esecuzione di una istruzione composta o l'attivazione di una procedura, e sia $f_i(n)$ il costo relativo all'esecuzione i-esima, i=1,2,...k. Il costo complessivo del programma è pari a

$$O(\Sigma_i f_i(n))$$

La dimostrazione della validità di quest'ultima regola è simile a quanto fatto precedentemente e viene omessa.

La regola 2 è molto utile nella valutazione di programmi ricorsivi. In questo caso $f_i(n)$ rappresenta il costo della i-esima attivazione della procedura. In molti casi il valore che denota il numero di ripetizioni è anch'esso una funzione delle dimensioni del problema, abbiamo cioè k=k(n). Non è difficile vedere che anche in questo caso la regola 2 risulta valida.

Un caso particolare di applicazione della regola 2 è quello in cui il programma richiede per k(n) volte l'esecuzione di un'istruzione composta o l'attivazione di una procedura avente ogni volta lo stesso costo $f(n)$; questo è il caso in cui le funzioni $f_i(n)$ sono tutte uguali a $f(n)$. In questo caso è facile osservare che il costo del programma è pari a O (k(n) $f(n)$).

2.3. Istruzione dominante

Introduciamo ora il concetto di istruzione dominante che permette, in molti casi, di semplificare in modo drastico la valutazione della complessità di un programma.

Definizione 3. Sia dato un programma o un algoritmo P il cui costo di esecuzione è $t(n)$. Una istruzione di P si dice istruzione o operazione *dominante* quando, per ogni intero n, essa viene eseguita, nel caso peggiore di *input* avente dimensione n, un numero di volte $d(n)$ che verifica la seguente condizione

$$t(n) < a\, d(n) + b$$

per opportune costanti a e b.

In altre parole, un'istruzione dominante viene eseguita un numero di volte proporzionale al costo di esecuzione di tutto l'algoritmo. È importante osservare che in un programma, più istruzioni possono essere dominanti, ma può anche accadere che il programma non contenga affatto istruzioni dominanti.

Regola 3. Supponiamo che un programma contenga un'istruzione dominante che, nel caso peggiore di input di dimensione n, viene eseguita $d(n)$ volte. La delimitazione superiore alla complessità del programma è $O(d(n))$.

La dimostrazione della correttezza della regola 3 è lasciata al lettore. Ci limitiamo ad osservare che questa semplificazione è possibile solo perché nella valutazione della complessità si trascurano costanti moltiplicative e termini additivi di ordine inferiore.

Per individuare un'istruzione dominante è sufficiente, in molti casi, esaminare le operazioni che sono contenute nei cicli più interni del programma. Ad esempio, nel caso della ricerca esaustiva considerarata nell'esempio precedente, è possibile valutare il costo del programma di fig.1 osservando che l'istruzione {3} è dominante. Infatti essa viene eseguita, nel caso peggiore, n volte; questo è sufficiente per dire che il programma ha costo lineare. Osserviamo inoltre che il test {4} del ciclo **repeat until** viene eseguito nel caso peggiore n volte e, quindi, rappresenta un'altra istruzione dominante.

2.4. Delimitazioni alla complessità di un problema

La notazione O esprime una delimitazione superiore alla complessità di un programma che risolve un particolare problema. Se un problema ha diversi algoritmi di soluzione con complessità diversa è naturale utilizzare l'algoritmo più efficiente per esprimere la complessità del problema.

Definizione 4. Un problema ha una *delimitazione superiore* O ($f(n)$) alla sua complessità, se esiste almeno un algoritmo di soluzione che ha complessità $O(f(n))$.

La ricerca di algoritmi sempre più efficienti ha permesso, in alcuni casi, di abbassare la complessità del problema da una funzione di tipo esponenziale ad una di tipo lineare. Può accadere, però, che questi miglioramenti non siano sempre possibili e che la delimitazione superiore alla complessità di un problema non possa essere abbassata oltre una certa soglia. Siamo perciò interessati a caratterizzare la delimitazione inferiore alla complessità del problema o, in altre parole, la complessità intrinseca del problema.

Definizione 5. Un problema ha *delimitazione inferiore* Ω ($g(n)$) alla sua complessità, se è possibile provare che ogni algoritmo di soluzione deve avere costo Ω ($g(n)$).

Dimostrare che un problema ha una delimitazione inferiore Ω ($g(n)$)alla sua complessità vuole dire che *ogni algoritmo di soluzione* per quel problema deve avere complessità almeno Ω ($g(n)$). Si noti che la definizione precedente fa riferimento non solo agli algoritmi noti, ma ad ogni possibile algoritmo progettato o ancora da progettare.

Sottolineiamo la differenza tra le due definizioni di delimitazione superiore ed inferiore alla complessità di un problema. Nella definizione 4 un problema ha costo O ($f(n)$) se esiste un algoritmo che, per ogni n e per ogni input di

dimensione n, impiega una quantità di risorse minore di $f(n)$. Invece nella definizione 5 un problema ha costo $\Omega(g(n))$ se per ogni algoritmo di soluzione esiste una sequenza infinita di istanze del problema aventi dimensioni crescenti, ciascuna delle quali richiede una quantità di risorse maggiore di $g(n)$.

Pertanto, se un problema ha costo $\Omega(g(n))$ non è detto che, per ogni istanza del problema di dimensione n, il costo di esecuzione di ogni algoritmo di risoluzione sia almeno c $g(n)$ (per un'opportuna costante c). Ad esempio, per stabilire che un problema ha una delimitazione $\Omega(n **2)$ non è necessario dimostrare che ogni algoritmo di soluzione ha costo almeno $\Omega(n **2)$ per ogni istanza di dimensione n; è, invece, sufficiente mostrare che, per ogni algoritmo di soluzione esiste una sequenza infinita di istanze "difficili" di dimensione crescente che richiedono una quantità di risorse pari almeno a $\Omega(n **2)$.

L'ultima definizione di questa sezione è quella di *algoritmo ottimale*. Intuitivamente un algoritmo è ottimale quando non può esistere un altro algoritmo avente complessità inferiore (a meno di un fattore moltiplicativo costante).

Definizione 6. Un algoritmo di soluzione di un problema P è *ottimale* quando l'algoritmo ha complessità $O(f(n))$ e la delimitazione inferiore alla complessità del problema P è $\Omega(f(n))$. In questo caso la notazione $\theta(f(n))$ denota la complessità del problema.

Esempio.Consideriamo il problema del calcolo del valore in un punto di un polinomio avente grado n. Supponiamo che i valori dei coefficienti del polinomio siano memorizzati in un **array** il cui tipo è così definito

type *vett*= **array** [*0..n*] **of** *real*

La componente i-esima dell'array contiene il valore del coefficiente di $x**i$ e la componente zero dell'array contiene il valore del termine noto del polinomio. In questo caso la dimensione dell'input è pari a n, grado del polinomio.

La prima soluzione che consideriamo è data in fig. 3. La valutazione del costo di esecuzione del programma precedente è semplificata se osserviamo che l'istruzione 6, posta all'interno di due cicli, è un'istruzione dominante. Quando la variabile i vale 1 il ciclo più interno viene eseguito una sola volta. Successivamente la variabile i assume il valore due e il ciclo più interno viene eseguito due volte. Analogamente quando la variabile i vale 3, 4, ... il ciclo più interno viene eseguito 3, 4, ... volte. Pertanto l'istruzione 6 viene eseguita

$$1 + 2 + 3 + ... + (n - 1) + n$$


```
function pol_1 ( var a : vett): real;

var  i, j : integer;
     x, y, val : real;
{x contiene il valore del punto in cui si vuole valutare il polinomio, y il valore
di una potenza di x, val il valore del polinomio nel punto x}

     begin
{1}  val := a[0];
{2}  readln (x);
{3}  for i := 1 to n
     do begin
{4}       y := 1;
{5}       for j := 1 to i
{6}       do y := y * x;
{7}       val := val + (a[i] * y)
     end;
{8}  pol_1 := val
     end;
```

Fig. 3 - Programma quadratico per la valutazione di un polinomio in un punto

volte. La somma precedente è una serie aritmetica il cui valore è $n(n + 1)/2$. Pertanto il costo di esecuzione del programma è $O(n **2)$.

La seconda soluzione che consideriamo evita di calcolare ogni volta la potenza di x elevato a i a partire da zero, ma utilizza i calcoli precedentemente fatti. A questo scopo è sufficiente memorizzare il valore di y e poi aggiornarlo all'interno del ciclo principale. La funzione è data in fig. 4.

È facile vedere che le istruzioni 5 e 6, contenute all'interno del ciclo, sono le istruzioni dominanti e vengono eseguite n volte ciascuna. Pertanto la complessità della procedura *pol_2* è $O(n)$. Il programma che abbiamo ottenuto è quindi più efficiente del programma *pol_1*. Osserviamo inoltre che l'algorimo *pol_2* è ottimale perché è possibile dimostrare che ogni algoritmo per il calcolo del valore in un punto di un polinomio di grado n, deve compiere almeno $O(n)$ operazioni.

```
function pol_2 ( var a : vett): real;

var i : integer;
    x, y, val : real;
{x contiene il valore del punto in cui si vuole valutare il polinomio, y il valore
di una potenza di x, val il valore del polinomio nel punto x}
      begin
{1}   val := a[0];
{2}   readln (x);
{3}   y := 1;
{4}   for i := 1 to n
      do begin
{5}         y := y * x;
{6}         val := val + (a[i] * y)
        end;
{7}   pol_2: = val
      end;
```

Fig. 4 - Programma lineare per la valutazione di un polinomio in un punto

3. LA GESTIONE DI UNA TAVOLA

In questo paragrafo consideriamo il problema della gestione di una tavola che è stato già studiato nel capitolo dedicato alle strutture di dati.

Consideriamo una tavola di elementi così definiti

```
type elem = record
              inf : tipo_informazione;
              chiave : integer
              end
```

La tavola è composta da n elementi, dove n rappresenta la dimensione dell'input. Nel seguito del paragrafo analizziamo la complessità di diversi metodi per la gestione di una tavola che sono stati introdotti nel cap. 3.

3.1. Gestione sequenziale

Il metodo di rappresentazione della tavola che dapprima consideriamo è quello mediante un array di record: ogni componente dell'array memorizza un elemento e gli elementi sono posti in ordine qualunque. La ricerca di un elemento nella tavola avviene con l'algoritmo di ricerca esaustiva, che abbiamo già analizzato nel paragrafo precedente, dimostrando che ha complessità $O(n)$. Osserviamo che $O(n)$ è anche il costo degli algoritmi di inserimento e di eliminazione di un elemento dalla tavola.

Nel caso che la tavola sia realizzata utilizzando record e puntatori, allora è facile vedere che il costo delle operazioni di ricerca e di eliminazione è ancora $O(n)$, perché nel caso peggiore sono necessari n confronti. Per la valutazione del costo dell'operazione di inserimento si distinguono due casi. Se è noto che l'elemento da inserire non è già presente nella tavola, allora è possibile inserirlo all'inizio della lista con un numero costante di operazioni. Nel caso che non sia noto se l'elemento è già presente, è necessario effettuare prima dell'inserimento una ricerca preliminare di costo $O(n)$.

3.2. Gestione sequenziale ordinata: la ricerca binaria

Come abbiamo visto nel cap. 3, par. 8.1, e nel cap. 6, par. 2.2, il metodo della ricerca binaria può essere utilizzato per ricercare elementi in una tavola ordinata.

La complessità della ricerca binaria dipende dal particolare elemento da individuare: se ricerchiamo l'elemento mediano allora è sufficiente un solo confronto, mentre in tutti gli altri casi dobbiamo eseguirne un numero maggiore. Non è difficile convincersi che la complessità dell'algoritmo è data dal numero di confronti effettuato e che il caso peggiore si verifica quando ricerchiamo un elemento non presente nella tavola.

Per valutare la complessità della procedura di ricerca binaria nel caso peggiore osserviamo che, dopo un confronto, dobbiamo proseguire la ricerca su metà degli elementi. Inoltre ogni successivo confronto permette di dimezzare ulteriormente la dimensione della tavola su cui proseguire la ricerca. Dopo h confronti la dimensione della tavola su cui dobbiamo continuare la ricerca è

$$n / (2^{**}h)$$

Pertanto, nel caso peggiore, il numero di confronti è il più piccolo intero m che soddisfa la seguente relazione