Corso di Laurea in INFORMATICA

a.a. 2012-2013

Algoritmi e Strutture Dati

MODULO 4

Complessità degli algoritmi ed efficienza dei programmi

Valutazione dell'efficienza dei programmi

Intuitivamente un programma è più efficiente di un altro se la sua esecuzione richiede meno risorse di calcolo.

Le risorse di calcolo che si considerano sono

- Il tempo di elaborazione richiesto
- La quantità di memoria necessaria

Se si potesse valutare il tempo di calcolo in unità standard come i secondi, la valutazione del costo di un programma sarebbe semplice: basterebbe eseguire il programma misurando il tempo necessario per l'esecuzione.

Valutazione dell'efficienza dei programmi

Il metodo proposto non è attendibile perchè bisogna considerare le condizioni in cui si effettuano le prove. Bisogna considerare:

- L'elaboratore su cui il programma viene eseguito
- Il compilatore usato per la traduzione
- La modalità di ingresso dei dati
- La significatività dei dati sui quali è eseguita la prova

Utilizzando unità di tempo standard come i secondi non otteniamo una valutazione oggettiva del costo di esecuzione di un programma

Complessità computazionale

L'efficienza di un programma è legata alla complessità dell'algoritmo.
La ***teoria della complessità*** studia in modo oggettivo l'uso delle ***risorse di calcolo*** necessarie per la computazione di algoritmi.
Per risolvere un problema spesso sono disponibili molti algoritmi diversi .

Come scegliere il migliore?

In genere si valuta *la bontà* di un algoritmo o si confrontano più algoritmi sulla base del comportamento che questi presentano al crescere della *dimensione* del problema.

Teoria della complessita'

Questo studio è importante al fine di:

- poter confrontare algoritmi diversi che risolvono lo stesso problema;
- stabilire se dei problemi ammettono algoritmi risolutivi che sono praticamente computabili.

Per i problemi computabili si parla di:

- ***Complessità dell'algoritmo**:* è una misura delle ***risorse di calcolo*** consumate durante la computazione.
- ***Efficienza dell'algoritmo (e del programma)**:* è inversamente proporzionale alla sua complessità.

Come valutare la complessità

Le ***risorse:*** TEMPO e SPAZIO DI MEMORIA

Quindi si parla di:

- COMPLESSITA' IN TEMPO
- COMPLESSITA' IN SPAZIO

di un algoritmo.

Per poter valutare la complessità occorre definire un

MODELLO DI COSTO

che dipende dal particolare modello di macchina astratta a cui si fa riferimento.

Come valutare la complessità

La ***risorsa*** **tempo**

Le unità di misura che si assumono in teoria della complessità non sono unità di tempo propriamente intese, per due ragioni:

1. Non ha senso parlare di tempi di esecuzione quando si considerano modelli di macchine astratte.
2. Anche se descrivessimo due algoritmi attraverso uno specifico linguaggio di programmazione disponibile per un calcolatore reale, il confronto dei tempi di esecuzione dei due programmi sarebbe inattendibile.

…..ma la complessità dipende anche dalla configurazione dei **dati in input**?

L'influenza della dimensione dell'input

Il costo di esecuzione di un programma dipende anche dai particolari dati di ingresso.

Ad esempio il costo di esecuzione di un programma di ordinamento di un insieme di numeri dipende dalle dimensioni dell'insieme che si considera.

Per tener conto del numero di dati con cui si esegue il programma assumiamo che la dimensione dell'input rappresenta *l'argomento della funzione che esprime il costo di esecuzione di un programma*.

La dimensione dell'input

Si può caratterizzare tale ***dimensione*** mediante un intero n che è precisamente identificato nella Macchina di Turing come la lunghezza della porzione di nastro che contiene i dati di ingresso.

Impiegando un elaboratore ed un suo linguaggio di programmazione, la ***dimensione n*** è *lo spazio occupato, nella memoria dell'elaboratore, dai dati relativi al problema da risolvere, o più in generale un numero proporzionale a questo spazio.*

Esempi:

- se si opera su matrici n sarà il numero dei suoi elementi
- se si opera su un insieme, n sarà il numero dei suoi elementi
- se si opera su un grafo, n sarà il numero dei nodi o il numero di archi o la somma dei due numeri

Complessità in tempo e complessità asintotica

Fissata la dimensione **n**, il tempo che un algoritmo impiega a risolvere il problema è la

complessità in tempo

Nostro obiettivo principale sarà esprimere la complessità in tempo come funzione di n e spesso ci limiteremo a studiare il comportamento di tale funzione al crescere di n (**complessità asintotica** o semplicemente complessità) considerando così i soli termini prevalenti e tralasciando a volte anche le costanti moltiplicative.

$$n \to \infty$$

Perché si studia la complessità asintotica

Lo studio della complessità asintotica è motivato dal fatto che gli algoritmi sono sempre definiti per **n** generico: se per valori piccoli di **n** due algoritmi possono avere efficienza confrontabile, è sempre quello che ha il termine massimo di grado più basso a richiedere minor tempo di esecuzione per un numero illimitato di valori di **n** superiori ad un opportuno valore $\mathbf{n_0}$.

Definire un modello di costo

L'analisi della efficienza di un programma e' basata sull'ipotesi che il costo di ogni istruzione semplice e di ogni operazione di confronto sia pari ad un'unita' di costo, indipendentemente dal linguaggio e dal sistema usato.

Ci limiteremo a **contare le operazioni eseguite** o alcune operazioni chiave o preminenti ammettendo che il tempo complessivo di esecuzione sia proporzionale al numero di tali operazioni.

Tratteremo spesso come non significative le costanti moltiplicative e studieremo le **funzioni di complessità** nel loro ordine di grandezza.

Il modello di costo

Il modello di costo che si utilizza è quello di un linguaggio di programmazione lineare.

Assumiamo che:

- Il costo di esecuzione di ogni ***istruzione semplice***
  - assegnazione ←
  - aritmetica +, -, *, /
  - logica AND, OR, NOT
  - confronto ≤, ≥, <, >, =, ≠
  - lettura/scrittura

  è **unitario.**
- Il costo di esecuzione di ***un'istruzione composta*** è pari alla somma dei costi delle istruzioni che la compongono

- Il costo di ***un'operazione di selezione***

  If *cond* **then** *S1* **else** *S2*

  è dato da:

  costo test *cond* + costo *S1* se cond

  costo test *cond* + costo *S2* se ¬ cond

- Il costo di ***un'istruzione di ciclo***: **while** *cond* **do** *S1*

  è dato da:

  costo test *cond* + (costo test *cond* + costo *S1*) * n

  se il ciclo è ripetuto n volte.

- Il costo di un ***ciclo "repeat"*** **repeat** *S1* **until** *cond*

  è dato da:

  (costo test *cond* + costo *S1*) * n

  se il ciclo è ripetuto n volte.

- Il costo di una istruzione ***do varying*** **for** *i*= 1 **to** *n* **do** *S1*

  è equivalente al costo del seguente ciclo

  $i \leftarrow 1$
  while $i \leq n$ **do**
  S1;
  $i \leftarrow i+1$

  ed è quindi dato da: 2 + (costo di *S1* + 3) * n

Tempo di calcolo di **min**()

- Ogni istruzione richiede un tempo costante per essere eseguita
- Costante diversa da istruzione a istruzione
- Ogni istruzione viene eseguita un certo numero di volte, dipendente da *n*

integer min(ITEM[ ] A, **integer** n)

	Costo	# Volte
ITEM $min \leftarrow A[1]$	c_1	1
for integer $i \leftarrow 2$ **to** n **do**	c_2	n
if $A[i] < min$ **then**	c_3	$n-1$
$min \leftarrow A[i]$	c_4	$n-1$
return min	c_5	1

Il costo ***T(n)*** di MIN si ottiene **sommando su tutte le istruzioni il prodotto del costo di ciascuna istruzione per il numero di volte che tale istruzione è eseguita**:

T(n) = C1 + C2 + C3 n + C4 (n - 1) + C5 (n - 1) + C6 =

=(C3 + C4 + C5) n + (C1 + C2 + C6 - C4 - C5).

Pertanto, il costo di MIN può essere espresso come

T(n) = an + b con a e b costanti intere positive.

Complessità in spazio

La complessità in spazio è il massimo spazio occupato in memoria durante l'esecuzione dell'algoritmo; questo, infatti, può costruire insiemi di dati intermedi o di servizio, oltre ad operare sui dati iniziali e finali.

Anche in questo caso ci si limita in genere allo studio della complessità asintotica.

Poiché abbiamo a disposizione memorie grandissime a basso costo, solitamente si privilegia lo studio della complessità in tempo.

Costo per lo spazio

Essendo n l'argomento della funzione che esprime il costo di esecuzione di un programma il costo è unario per i dati di tipo semplice mentre è dato da:

n * dimensione di un elemento per insiemi, array di n elementi etc.

Il costo di un record è dato dalla somma dei costi delle singole componenti.

Esempio:

L'algoritmo

somma $\leftarrow$ *0*

for *i* =1 **to** *n* **do** *somma* $\leftarrow$ *somma + A [i]*

ha **complessità in spazio** pari a 13 se n = 10

Funzioni di complessità

***RIASSUMENDO*:**

La complessità di un algoritmo è funzione della dimensione dei dati, ovvero della mole dei dati del problema da risolvere.

L'individuazione della dimensione dei dati è per lo più immediata.

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 dei dati, n.

Le funzioni di complessità sono caratterizzate da due proprietà:

1. Assumono solo valori positivi
2. Sono crescenti rispetto alla dimensione dei dati.

Complessità e configurazioni

La complessità di un algoritmo non può sempre essere caratterizzata da una sola funzione di complessità.

A parità di dimensione di dati, il tempo di esecuzione può dipendere dalla specifica configurazione dei dati.

Si considerano di solito tre differenti tipi di complessità: complessità nel caso *medio*, *ottimo* e *pessimo*.

Complessità media

- Valore della complessità di un algoritmo, mediato su tutte le possibili occorrenze iniziali dei dati.

- Si usa spesso la probabilità.

- Il calcolo è spesso difficile.

Complessità nel caso ottimo

- Si ottiene considerando, a parità di dimensione dei dati, la configurazione che dà luogo al minimo tempo di esecuzione.

- Tale complessità è per lo più di interesse secondario anche se è abbastanza facile da determinare.

Complessità nel caso pessimo

- Si intende la complessità relativa a quella particolare occorrenza iniziale dei dati per cui l'algoritmo ha comportamento pessimo.

- Tale funzione di complessità fornisce un **limite superiore** alla complessità, entro cui il funzionamento dell'algoritmo è sempre garantito.

Complessità e configurazioni

- ***CASO PESSIMO:*** si ottiene considerando quella particolare configurazione che, a parità di dimensione dei dati, dà luogo al massimo tempo di calcolo. La corrispondente funzione di complessità è indicata con $f_{pess}(n)$.
- ***CASO OTTIMO:*** si ottiene considerando la configurazione che dà luogo al minimo tempo di calcolo. La corrispondente funzione di complessità è indicata con $f_{ott}(n)$.
- ***CASO MEDIO:*** si riferisce al tempo di calcolo mediato su tutte le possibili configurazioni dei dati, sempre per una data dimensione n. Funzione di complessità: $f_{med}(n)$.

Esistono algoritmi in cui tutte le possibili configurazioni dei dati sono equivalenti per quanto riguarda la complessita' in tempo (coincidono complessita' ottima, pessima e media).

Esempi:

$f_{pess} = f_{ott} = f_{med}$ per l'algoritmo del fattoriale.

$f_{pess} = f_{ott} = f_{med}$ per l'algoritmo del Min di n elementi.

Esempio: Ricerca esaustiva 1/3

ricerca_esaustiva (t : tipotavola;
k : tipochiave; 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 */

{1} i ← 0
{2} **repeat**
{3} i ← i + 1
{4} **until** (t [i].chiave = k) **or** (i = n)
{5} **if** (t [i].chiave = k) **then**
{6} trovato ← true
{7} **else** trovato ← false

Esempio: Ricerca esaustiva 2/3

E' importante la posizione del particolare elemento che si vuole individuare: se l'elemento cercato è il primo della tavola allora si effettua un solo confronto (questa situazione rappresena il ***caso ottimo***); 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, perchè 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

$$1 + n + n + 1 + 1 = 3 + 2n$$

Esempio: Ricerca esaustiva 3/3

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

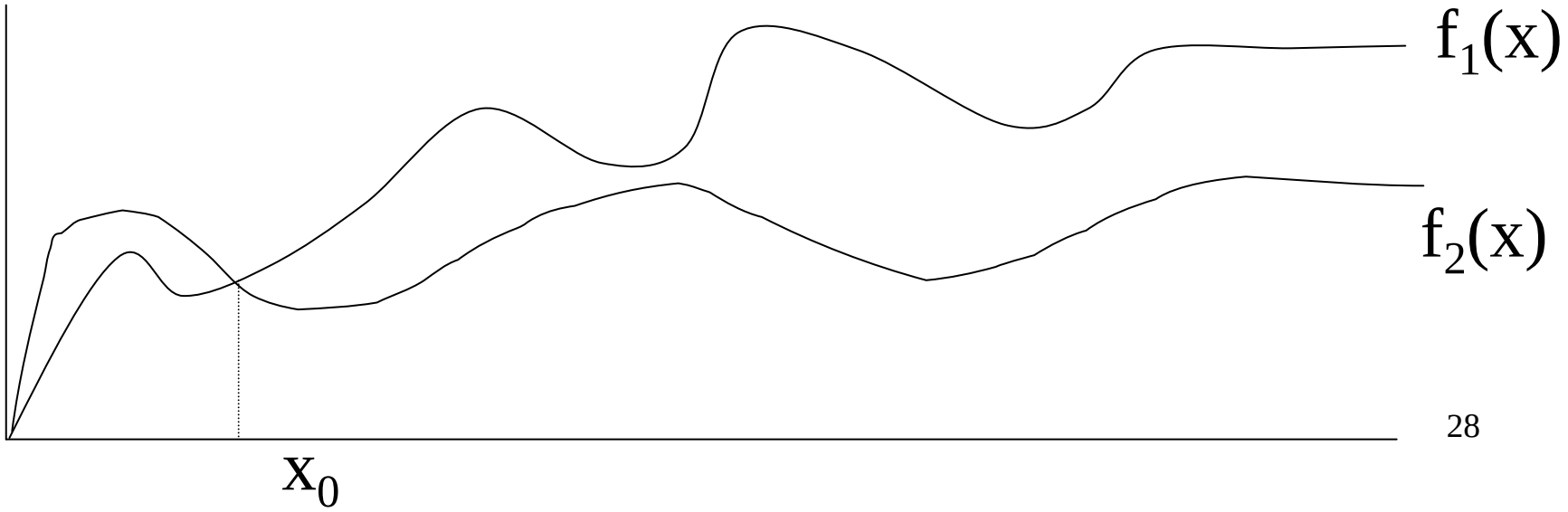
Se si vuole valutare il comportamento del programma nel ***caso medio*** è necessario distinguere il caso di ricerca con successo da quello di ricerca infruttuosa. Nel caso di ricerca fruttuosa, se si assume che tutti gli elementi dell'array possano essere ricercati con uguale probabilità pari a 1/n, 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:

$$\sum_{1 \le i \le n} \text{Prob}(E(i))\, i = \sum_{1 \le i \le n} i/n = (n + 1)/2$$

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

Comportamento asintotico

La complessità asintotica è una funzione che esprime la dipendenza del tempo di esecuzione al crescere della dimensione dei dati. Per semplificare l'analisi, si ritiene sufficiente stabilire il ***comportamento asintotico*** quando le dimensioni dell'input tendono all'infinito. Si trascurano cosi' costanti moltiplicative e termini additivi di ordine inferiore.



Comportamento asintotico

Due programmi i cui costi sono espressi rispettivamente dalle funzioni

(3 + n) e (100n + 3027)

sono caratterizzati dalla **stessa complessità asintotica**.

(3+n) esprime un costo minore della funzione (100n + 3027) e, quindi, ignorare la costante moltiplicativa può essere in alcuni casi una semplificazione eccessiva.

Però, 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.

Esempi

Esempio

A_1: $f_1(n) = 3n^2 - 4n + 2$ A_2: $f_2(n) = 2n + 3$

$n = 1, 2$ $f_1(n) < f_2(n)$

$n = 3, 4$ $f_1(n) > f_2(n)$

Qual è l'algoritmo più efficiente?

A_2, perché $\forall\ n > 2$ $f_1(n) > f_2(n)$

Esempio

Le 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

Delimitazioni della complessità

Il costo di esecuzione di un programma, o di un algoritmo, si esprime come funzione delle dimensioni dell'input in cui si ignorano le costanti moltiplicative.

Nello studio asintotico vengono trascurate le costanti moltiplicative e i termini la cui crescita asintotica e' inferiore a quella di altri termini.

Si definiscono inoltre diversi tipi di **delimitazioni** che aiutano a formalizzare la complessità di un programma (o di un algoritmo).

Notazioni

O(f(n)) è l'insieme di tutte le funzioni g(n) tali che esistono due costanti positive c e m per cui g(n)<=cf(n) per ogni n>= m

g(n)$\in$O(f(n)) tradizionalmente si legge "g(n) è di ordine f(n)" e fornisce un limite superiore al comportamento asintotico della funzione g ovvero g non si comporta asintoticamente peggio di f.

Applicata alla funzione di complessità, la notazione O ne delimita superiormente la crescita e fornisce un indicatore di bontà dell'algoritmo.

Esempio
La funzione:
$f(n) = 3n^2 + 3n - 1$
è un $O(n^2)$ dal momento che esistono due costanti c = 4 ed m = 3 tali che

$$\forall n \geq 3 \quad f(n) \leq 4n^2.$$

In generale, $f(n)$ è un $O(n^k)$, $k \geq 2$

Notazioni

Ω(f(n)) è l'insieme di tutte le funzioni g(n) tali che esistono due costanti positive c e m per cui g(n)>=cf(n) per ogni n>= m.

g(n)∈Ω(f(n)) tradizionalmente si legge "g(n) è di ordine Ω f(n)" e fornisce un limite inferiore al comportamento asintotico della funzione.

Notazioni

θ(f(n)) è l'insieme di tutte le funzioni g(n) che sono sia Ω f(n) sia Of(n). Formalmente è l'insieme di tutte le funzioni g(n) tali che esistano tre costanti positive c,d, ed m per cui df(n)>=g(n)>=cf(n)

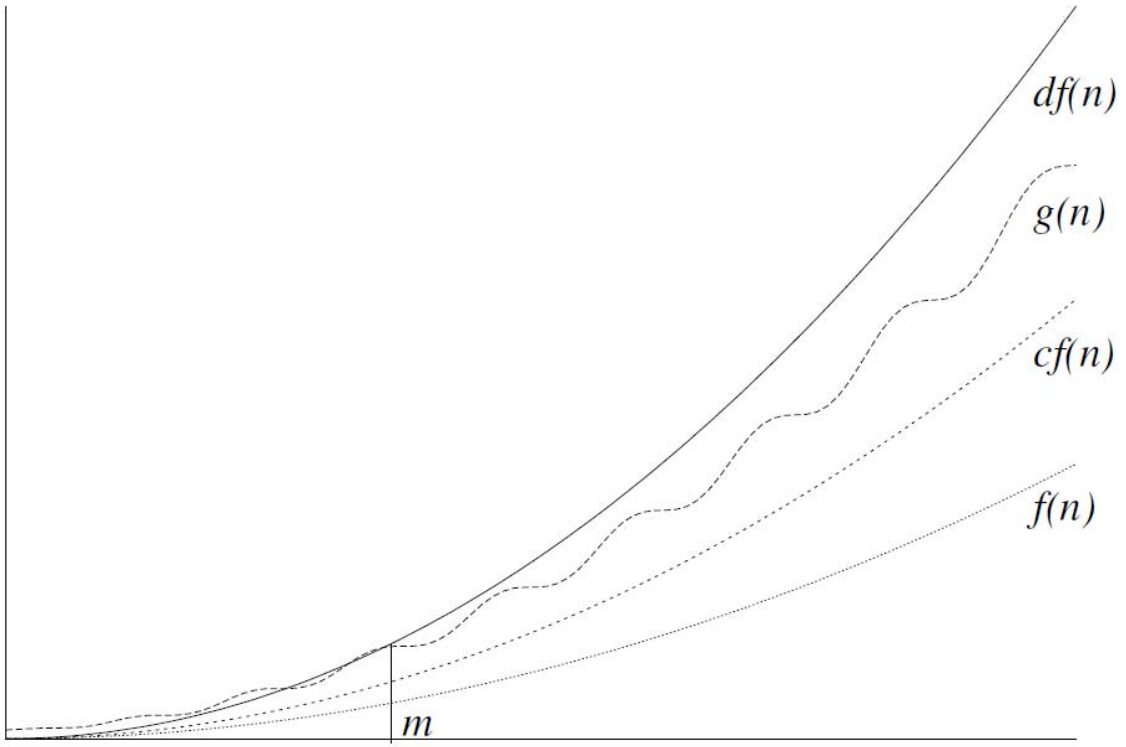
g(n)$\in\theta$(f(n)) tradizionalmente si legge "g(n) è di ordine θf(n)"; la g si comporta asintoticamente esattamente come la f, per cui l'andamento di f caratterizza precisamente quello di g.

Proprietà di O, Ω e θ

- $g \in O(f)$ implica $(f+g) \in O(f)$
- $f_1 \in O(g_1)$, $f_2 \in O(g_2)$ implica $f_1 + f_2 \in O(g_1+g_2)$
- O e Ω sono relazioni riflessive e transitive
- θ è una relazione di equivalenza
- $f \in O(g)$ se e solo se $g \in \Omega(f)$

Limiti asintotici superiori e inferiori

- $g(n) \in O(f(n)) \Leftrightarrow \exists c > 0, m > 0 : g(n) \leq cf(n), \forall n \geq m$
- $g(n) \in \Omega(f(n)) \Leftrightarrow \exists c > 0, m > 0 : cf(n) \leq g(n), \forall n \geq m$
- $g(n) \in \Theta(f(n)) \Leftrightarrow \exists c > 0, d > 0, m > 0 : cf(n) \leq g(n) \leq d(f(n)), \forall n \geq m$



Valutazione della complessità di un programma

Diamo una serie di regole che aiutano a trovare la delimitazione superiore della complessita'.

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

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

Valutazione della complessità di un programma

Inoltre si noti che

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

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

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

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

Valutazione della complessità di un programma

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

$$O\left(\sum_i f_i\ (n)\right)$$

Istruzione dominante 1/2

Il concetto di istruzione dominante permette, in molti casi, di semplificare in modo drastico la valutazione della complessità di un programma.

Definizione

Sia dato un programma P il cui costo di esecuzione è t(n). Una istruzione di P si dice istruzione o operazione *dominante* se, 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.

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

Istruzione dominante 2/2

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)).

Per individuare un'istruzione dominante è sufficiente, spesso, esaminare le operazioni contenute nei cicli più interni del programma.

Ad esempio, nel caso della ricerca esaustiva considerata in un esempio precedente, è possibile valutare il costo del programma mostrato osservando che l'istruzione {3} è dominante perché 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.

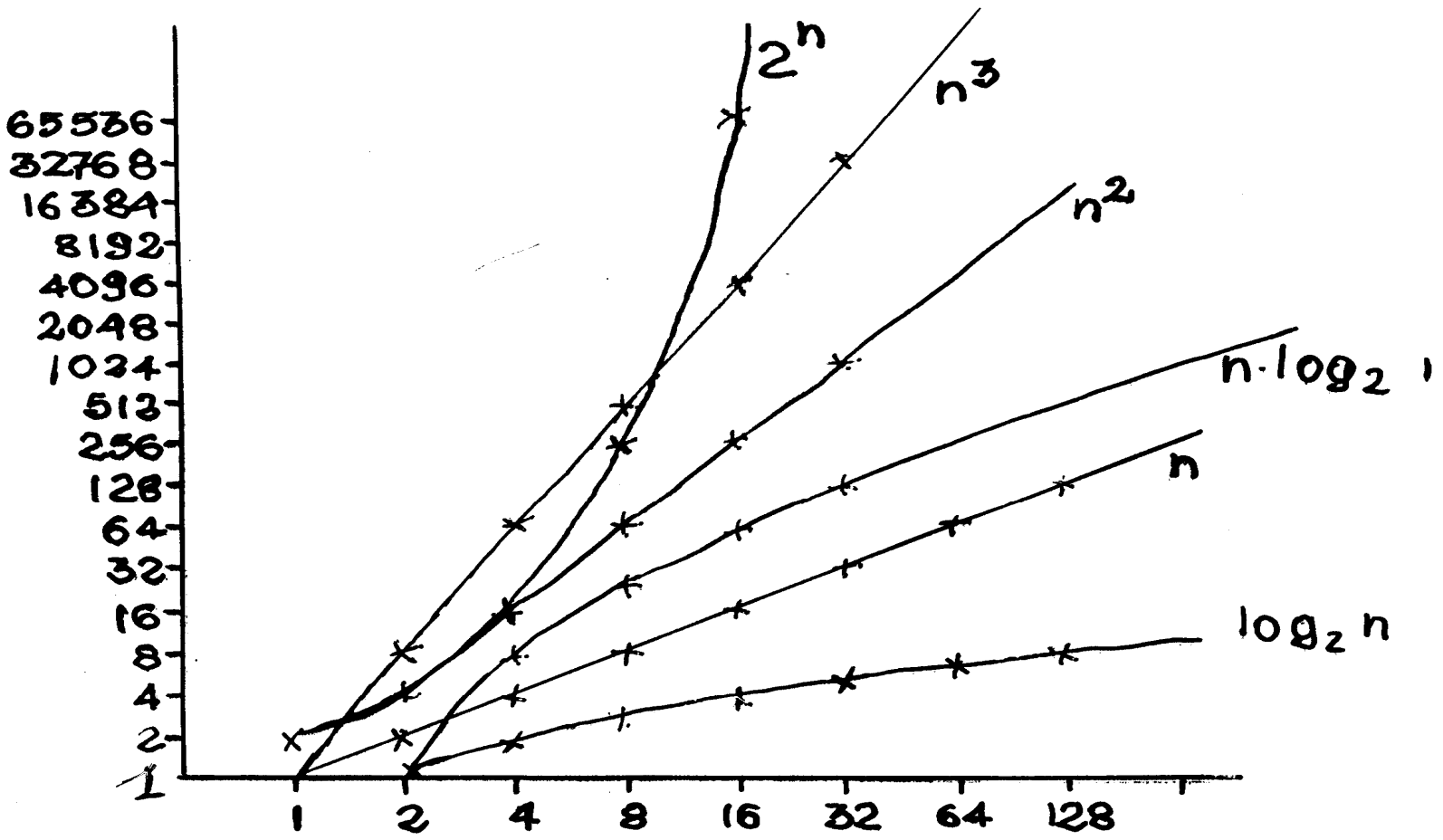
Classi di complessità computazionale

Grazie all'analisi della complessità asintotica possiamo classificare gli algoritmi/programmi nelle seguenti classi di complessità:

costante	O(1)
logaritmica	O (logn)
lineare	O (n)
nlog	O (nlogn)
quadratica	O (n^2)
cubica	O (n^3)
esponenziale	O (a^n) 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.

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



$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < \ldots < O(2^n)$$

Un esempio:Ordinare un array

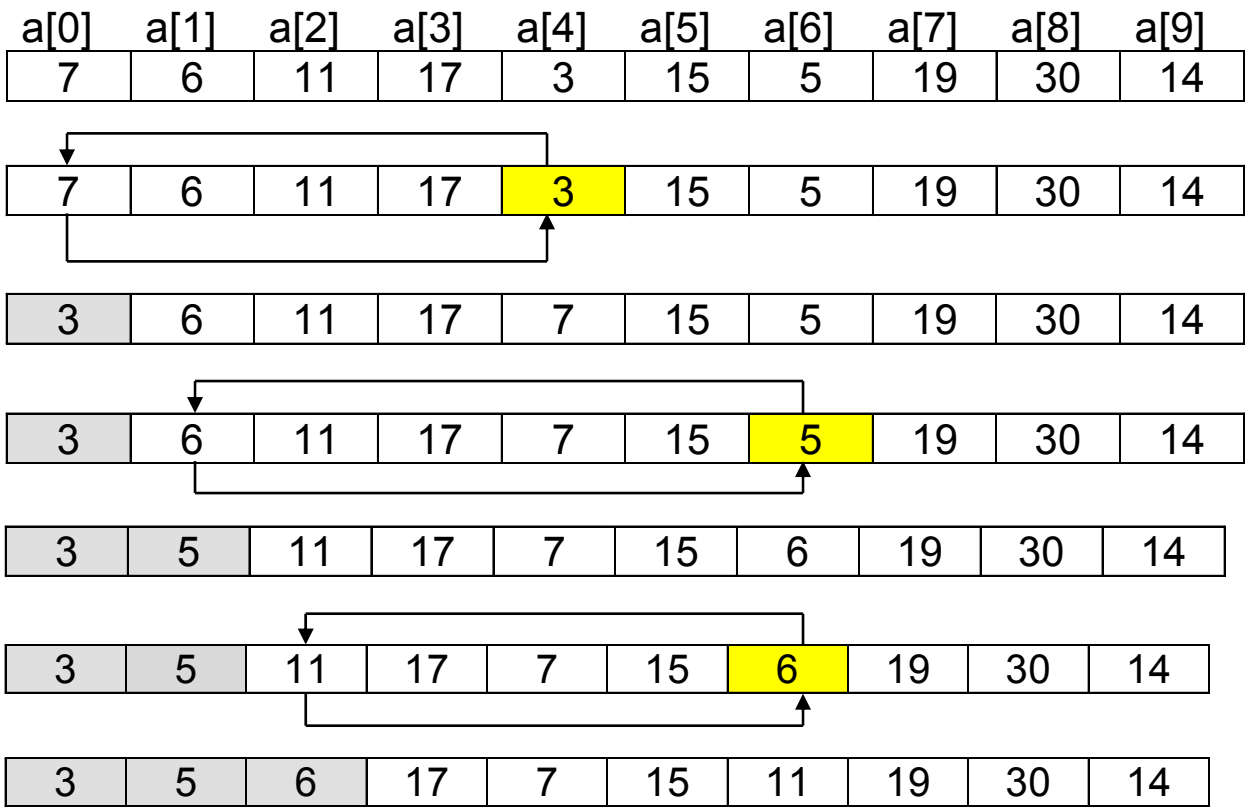
- Ordinare un elenco di elementi è un compito molto frequente
  - ordinare numeri in modo crescente
  - ordinare numeri in modo decrescente
  - ordinare stringhe in modo alfabetico
- Vi sono molti modi per ordinare un elenco *Selection sort*
  - uno dei più facili
  - non il più efficiente, ma facile da capire e da programmare

Algoritmo Selection Sort

Per ordinare un array di interi in modo crescente:

- cerca nell'array il numero più piccolo e scambialo con il primo elemento dell'array
  - la parte ordinata dell'array è ora il primo elemento, mentre quella non ancora ordinata sono i rimanenti elementi
- cerca nella parte non ordinata il numero più piccolo e scambialo con il secondo elemento dell'array
- ripeti la ricerca e lo scambio fino a quando tutti gli elementi sono al posto giusto
  - ogni iterazione aumenta di 1 la lunghezza della parte ordinata e diminuisce di 1 quella della parte non ordinata

Esempio


a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]
7 6 11 17 3 15 5 19 30 14
7 6 11 17 3 15 5 19 30 14
3 6 11 17 7 15 5 19 30 14
3 6 11 17 7 15 5 19 30 14
3 5 11 17 7 15 6 19 30 14
3 5 11 17 7 15 6 19 30 14
3 5 6 17 7 15 11 19 30 14

selection sort in C

```
void selezione(int v[], int n)
{
int a,b,c,scambio,t,j;
for(a=0; a<n-1;a++){
        scambio=0;
        c=a;
        t=v[a];
        for(b=a+1;b<n;b++){
            if(v[b]<t){
              c=b;
              t=v[b];
              scambio=1;
              }
         }
         if(scambio){
               v[c]=v[a];
               v[a]=t;
         }
     }
}
```

Complessità del Selection sort

L'ordinamento per selezione ha un ciclo più esterno e uno più interno con proprietà simili, sebbene con scopi differenti. Il ciclo più esterno viene eseguito una volta per ogni valore nell'array e quello più interno confronta il valore scelto dal ciclo più esterno con molti, se non tutti, i valori rimanenti nell'array. Quindi esegue n^2 confronti ove n è il valore di elementi dell'array. ***Selection sort è quindi di ordine n^2***.

Algoritmo di ordinamento bubblesort

Idea:

- leggere l'array v[0],…,v[n-1], da sinistra a destra, confrontando tra di loro ogni coppia di elementi contigui, e, se questi non sono in ordine tra di loro, scambiarli di posto;
- a fine lettura l'elemento massimo si troverà al posto n-1;*
- se non si sono effettuati scambi, l'array è già ordinato, altrimenti ordinare con lo stesso metodo l'array v[0],…,v[n-2].

*

Infatti ad ogni confronto, qualunque elemento segua il massimo sarà minore di questo, e quindi verrà scambiato.

bubblesort in C

```
void BubbleSort(unsigned int v[],int n){
    int i=-1,t,another;

            do {
        another=0;
        for (i=1;i<n;i++){
            if ((v[i-1]>v[i])) {
                t=v[i-1];
                 v[b-1]=v[b];
                 v[b]=t;
                 another=1;}
                 n=n-1;
            }
    } while (another);
}
```

questo ciclo ogni volta confronta tutte le coppie di elementi consecutivi, e viene eseguito n volte alla prima esecuzione del ciclo esterno, (n-1) alla seconda, e così via.

questo ciclo viene eseguito finchè non vengono più effettuati scambi.

Complessità di bubblesort (numero di scambi)

Caso migliore (l'array è già ordinato): si fa un solo ciclo esterno, senza che siano effettuati scambi, e quindi vengono fatti (n-1) confronti, quindi:

$C_{bubblesort}(n) \in O(n)$;

Caso peggiore (l'array è ordinato in ordine inverso): ad ogni ciclo interno l'elemento massimo viene posto nella sua posizione finale, mentre il resto dell'array rimane ordinato in ordine inverso.

Si effettua uno scambio ad ogni ciclo, cioe':

$C_{bubblesort}(n) = (n-1)+(n-2)+\ldots+1 \in O(n^2)$

Caso medio: come il caso peggiore

Scomposizione dei costi di un programma

Per valutare il costo di un programma a partire dal costo delle sue componenti va affrontato il problema del costo di programmi con sottoprogrammi ripetuti.
Sia A un programma la cui esecuzione consiste nella esecuzione ripetuta di chiamate di funzione il cui costo all'i-esima ripetizione è $f_i(n)$ dove n è la dimensione dell'input di A. Sia g(n) il numero di ripetizioni legate alla dimensione dell'input. Il costo del programma $f_a(n)$ è dato da

$$O\left(\sum_{i=1}^{g(n)} f_i(n)\right)$$

Scomposizione dei costi di un programma

Esempio
Si consideri la funzione in grado di determinare se tutti gli elementi di un vettore w siano presenti in un altro vettore v (banale ricerca sequenziale).

```
Bool CercaInTutti (int v[], int w[])
// v e w sono vettori di N interi. Se tutti gli elementi di w sono in v
//  la funzione restituisce true, altrimenti restituisce false
{ for (int i= 0; i<N; i++)
    if (!RicercaSequenziale(v, w[i]))
       return false;
    return true;
}
```

Il numero massimo di chiamate al sottoprogramma RicercaSequenziale () è pari ad N e si ha quando tutti gli elementi di w vengono trovati in v. Poiché il costo dell'esecuzione del sottoprogramma è delimitato superiormente da O(N) si può concludere che CercaInTutti() è delimitato superiormente da $O(N^2)$

Costi di un programma dovuti a operatori su ADT

Da quanto detto si deduce che, poiché gli operatori sulle strutture dati definite da noi sono, di fatto, sottoprogrammi che hanno una loro complessità e vengono applicati più volte, il costo di una realizzazione piuttosto che un'altra incide sicuramente sul costo complessivo del programma che utilizza quegli operatori.

Ad esempio, la complessità degli operatori su lista realizzati con vettore sequenziale è O(1) per tutti gli operatori tranne che per quelli di modifica della struttura (O(n)).

La complessità degli operatori su liste bidirezionali realizzate con cursori/puntatori è costante O(1) per tutti gli operatori e l'occupazione di memoria è O(maxlung) per tutte le liste. Per le realizzazioni con cursori va anche considerata **INIZIALIZZALISTALIBERA.**

REALIZZAZIONE SEQUENZIALE CON VETTORE

Possibile dichiarazione di tipo

```
tipo_atomi = integer
posizione = 0..nmax_lista
tipo_lista =    RECORD
    elementi : array [1..nmax_lista] of tipo_atomi
    primo, lunghezza : posizione
```

```
FUNCTION LISTAVUOTA(LIS di tipo lista)→boolean     {O(1)}
    LISTAVUOTA←(LIS.LUNGHEZZA = 0)
```

```
FUNCTION PRIMOLISTA(LIS di tipo_lista) → posizione {O(1)}
    if LISTAVUOTA(LIS)
        then writeln('OPERAZIONE NON ESEGUIBILE')
    else
        PRIMOLISTA←LIS.PRIMO
```

```
PROCEDURE CANCLISTA (p di tipo posizione, LIS di tipo LISTA,
                     i di tipo posizione);
                                                          {O(n)}
  begin
  if(p<LIS.PRIMO) or (p>LIS.LUNGHEZZA) then
  writeln('POSIZIONE',p,'ERRATA PER CANCELLA')
  else
      for i=p to LIS.LUNGHEZZA – 1 do
          LIS.ELEMENTI [ i ]←LIS.ELEMENTI [ i+1];
          LIS.LUNGHEZZA←LIS.LUNGHEZZA – 1;
      end;
  end;
```

NELLA REALIZZAZIONE DI LISTA MONODIREZIONALE CON PUNTATORI TUTTI GLI OPERATORI HANNO COMPLESSITA' O(1). L'UNICA ECCEZIONE E' PREDLISTA CHE HA COMPLESSITA' O(n)

```
FUNCTION PREDLISTA (p di tipo posizione,  L di tipo lista) → posizione
    IF p ≠ nil THEN
        PREDLISTA←nil
    ELSE
        q←L
        WHILE q^.successivo ≠ p DO
        q←q^.successivo
        PREDLISTA←q
```

Complessità della fusione di liste 1/4

La complessità della procedura fusione dipende dalla complessità degli operatori su lista utilizzati, ovvero:

- CREALISTA
- PRIMOLISTA
- FINELISTA
- LEGGILISTA
- INSLISTA
- SUCCLISTA

Nella realizzazione con puntatori e doppi puntatori la complessità di questi operatori è O(1). Nel caso di realizzazione con cursori occorre tenere presente il costo della procedura INIZIALIZZALISTALIBERA che comunque è eseguita fuori della procedura fusione e quindi non va conteggiato a fini della complessità della procedura.

```
fusione(Lista1 di tipo Lista, Lista2, Lista3 : di tipo Lista)
    crealista(Lista3)
    p1 ← primolista(Lista1)
    p2 ← primolista(Lista2)
    p3 ← primolista(Lista3)
    while (not finelista(p1, Lista1) and not finelista (p2, Lista2)) do
        elem1 ← leggilista(p1, Lista1)
        elem2 ← leggilista(p2, Lista2)
        if elem1 < elem2 then
            inslista(elem1, p3, Lista3)
            p1 ← succlista (p1, Lista1)
        else
            inslista(elem2, p3, Lista3)
            p2 ← succlista (p2, Lista2)
        p3 ← succlista (p3, Lista3)
    while not finelista (p1, Lista1) do
        inslista (leggilista(p1, Lista1), p3, Lista3)
        p1 ← succlista (p1, Lista1)
        p3 := succlista (p3, Lista3)
    while not finelista (p2, Lista2) do
        inslista (leggilista(p2, Lista2), p3, Lista3)
        p2 ← succlista (p2, Lista2)
        p3 ← succlista (p3, Lista3)
```

Complessità della fusione di liste 2/4

Consideriamo come operazione dominante il confronto di elem1 con elem2. Allora la complessità dipende dalla configurazione dei dati di ingresso.

In particolare, dette m ed n le lunghezze rispettivamente di Lista1 e Lista2, possiamo distinguere i seguenti casi:

- CASO OTTIMO:

  tutti gli elementi della lista più corta sono minori degli elementi della lista più lunga. Si effettueranno tanti confronti quanti sono gli elementi della lista più corta:

$$f_{ott}(m, n) = \min\{m, n\}$$

Complessità della fusione di liste 3/4

- CASO PESSIMO:
  l'inserimento dell'ultimo elemento di una delle due liste può avvenire solo dopo l'inserimento del penultimo elemento dell'altra lista. Il numero di confronti sarà dato da:
  $f_{pess}(m, n) = m + n - 1$
- CASO MEDIO:
  senza perdere di generalità supponiamo essere m <= n.
  Consideriamo allora i seguenti eventi:
  $E_k \equiv$ l'ultimo elemento di Lista1 è inserito prima del K-esimo elemento di Lista2
  per K = 1,2,3,…, n.

Supponiamo che i vari eventi siano tutti equiprobabili (poiché non abbiamo informazioni sufficienti per sostenere il contrario, ipotizziamo una distribuzione uniforme), allora:

$$P(E_k) = 1/n$$

poiché i diversi eventi sono n.

Complessità della fusione di liste 4/4

Il costo, espresso in termini di numero di confronti, associato ad ogni evento è dato da:

$$costo\,(E_k) = m + k - 1$$

Si noti che E_1 corrisponde al caso ottimo mentre E_n al caso pessimo.

Allora la complessità nel caso medio sarà dato da:

$$f_{med}(m, n) = \sum_{k=1}^{n} P(E_k)\, costo\,(E_k) = 1/n \sum_{k=1}^{n} (m + k - 1) =$$

$$= 1/n \left[\sum_{k=1}^{n} m + \sum_{k=1}^{n} k - \sum_{k=1}^{n} 1 \right] = 1/n\,(m\,n + (n(n+1)/2) - n) =$$

$$m + (n+1)/2 - 1$$

Complessità dell'ordinamento di lista con NMS

Osserviamo che per una lista di n elementi, il numero massimo di passi di distribuzione e fusione delle catene è al più uguale a

$[\log_2 n] + 1$ in quanto ad ogni passo il numero di catene, che nel caso peggiore coincide con n stesso, viene dimezzato.
Quindi la complessità dipende dalla particolare configurazione dei dati in ingresso.
Consideriamo, come operazione dominante, il confronto

leggilista (pa, A) < leggilista (pb, B)

nella procedura fondiCatena.

Allora nel caso pessimo abbiamo $(\log_2 n)$ esecuzioni della procedura merge che ha complessità proporzionale a n, quindi:

$f_{pess}(n)$ è un $O(n \log_2 n)$