Ricorrenze e stima delle prestazioni

nell'analisi degli algoritmi

Lezione 5 di Programmazione 2

Docente: Giuseppe Scollo

Università di Catania, sede di Comiso (RG)
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Studi in Informatica applicata, AA 2006-7

Indice

relazioni di ricorrenza

molti algoritmi, progettati seconda la regola **"divide et impera"**, risolvono il problema decomponendolo in istanze più piccole dello stesso problema

più precisamente, la soluzione per un valore **N** del parametro primario (dimensione dell'input o di una struttura dati fondamentale dell'algoritmo) è espressa in termini della stessa soluzione per un valore minore di tale parametro

l'analisi delle prestazioni di tali algoritmi si basa sulla soluzione di **relazioni di ricorrenza**, nelle quali il costo computazionale C_N è definito induttivamente, a partire da un costo di base C_1, per il tramite di un'**equazione ricorsiva**

mostriamo nel seguito una semplice tecnica di soluzione di ricorrenze attraverso **sostituzioni successive** per alcune classi fondamentali di algoritmi

N.B. tecniche più rigorose saranno oggetto di studi più avanzati

C(N) = C(N-1) + N, per N > 1, C(1)=1

classe degli algoritmi: si esamina tutto l'input e se ne elimina un elemento, reiterando quindi l'algoritmo sul resto dell'input

ricorrenza: $C_N = C_{N-1} + N$, per $N > 1$, con $C_1 = 1$
soluzione: C_N

$= C_{N-1} + N$

$= C_{N-2} + (N-1) + N$

$= C_{N-3} + (N-2) + (N-1) + N$

$= \ldots$

$= C_1 + 2 + \ldots + (N-1) + N$

$= N(N+1)/2$

$\approx N^2/2$

C(N) = C(N/2) + 1, per N > 1, C(1)=1

classe degli algoritmi: si dimezza l'input ad ogni passo, di costo costante, reiterando quindi l'algoritmo sull'input dimezzato

ricorrenza: $C_N = C_{N/2} + 1$, per $N > 1$, con $C_1 = 1$

soluzione: per $N = 2^n$ si ha: C_{2^n}

$= C_{2^{n-1}} + 1$

$= C_{2^{n-2}} + 2$

$= C_{2^{n-3}} + 3$

$= \ldots$

$= C_1 + n$

$\approx n$

e più in generale $C_N \approx \lg N$ se si interpreta il dimezzamento come quoziente intero, $N/2 = \lfloor N/2 \rfloor$, poiché in tal caso il numero di iterazioni è uguale al numero di bit necessari alla rappresentazione binaria naturale di N, cioè $\lceil \lg N \rceil$

C(N) = C(N/2) + N, per N > 1, C(1)=0

classe degli algoritmi: ad ogni passo si esamina tutto l'input e lo si dimezza, reiterando quindi l'algoritmo sull'input dimezzato

ricorrenza: $C_N = C_{N/2} + N$, per $N > 1$, con $C_1 = 0$

soluzione: se si adopera il quoziente intero, $N/2^k = \lfloor N/2^k \rfloor$, si ha: C_N

$= C_{N/2} + N$

$= C_{N/4} + N/2 + N$

$= C_{N/8} + N/4 + N/2 + N$

$= \ldots$

$< N(1 + 1/2 + 1/4 + \ldots) = N(1/(1 - 1/2)) = 2N$

$\approx 2N$

dove il fattore 2 è il limite della serie geometrica di ragione 1/2, e l'errore di tale approssimazione per eccesso tende a 0 per $N \to \infty$

C(N) = 2C(N/2) + N, per N > 1, C(1)=0

classe degli algoritmi: ad ogni passo si esamina tutto l'input e lo si divide in due metà, reiterando quindi l'algoritmo su entrambe le metà

ricorrenza: $C_N = 2C_{N/2} + N$, per $N > 1$, con $C_1 = 0$
soluzione: nell'ipotesi che $N = 2^n$, si ha: $C_{2^n} = 2C_{2^{n-1}} + 2^n$
quindi, dividendo entrambi i membri per 2^n: $C_{2^n}/2^n$

$= C_{2^{n-1}}/2^{n-1} + 1$

$= C_{2^{n-2}}/2^{n-2} + 2$

$= \ldots$

$= C_1 + n$

$= n$

dunque $C_N = N \lg N$ nell'ipotesi posta, e più in generale $C_N \approx N \lg N$

C(N) = 2C(N/2) + 1, per N > 1, C(1)=1

classe degli algoritmi: ad ogni passo, di costo costante, si divide l'input in due metà, reiterando quindi l'algoritmo su entrambe le metà

ricorrenza: $C_N = 2C_{N/2} + 1$, per $N > 1$, con $C_1 = 1$
soluzione: nell'ipotesi che $N = 2^n$, si ha: $C_{2^n} = 2C_{2^{n-1}} + 1$
quindi, dividendo entrambi i membri per 2^n: $C_{2^n}/2^n$

$= C_{2^{n-1}}/2^{n-1} + 2^{-n}$

$= C_{2^{n-2}}/2^{n-2} + 2^{-(n-1)} + 2^{-n}$

$= \ldots$

$= C_1 + 2^{-1} + \ldots + 2^{-(n-1)} + 2^{-n}$

< 2

approssimando per eccesso con il limite della serie geometrica di ragione 1/2
e quindi, più generalmente per qualsiasi N : $C_N \approx 2N$

ricerca sequenziale

a titolo di esempio, vediamo l'uso di relazioni di ricorrenza nel confronto delle prestazioni di algoritmi diversi per uno stesso problema:

la ricerca di un elemento in un insieme

sia **N** la cardinalità dell'insieme dato, parametro primario dell'analisi prestazionale

assumiamo, senza perdita di generalità, che l'insieme dato sia memorizzato in un array, ovvero in una struttura dati ad accesso diretto

le prestazioni dell'algoritmo per questo problema risultano critiche per l'efficienza di algoritmi che debbano effettuare molte volte la ricerca in questione, ad esempio per un input costituito da **M** elementi da cercare, con **M >> N**

un semplice algoritmo per il problema in esame effettua la scansione sequenziale degli elementi nell'array, confrontandoli con l'elemento dato fino a quando

- l'elemento viene trovato: **esito positivo**
- si esaurisce lo spazio di ricerca: **esito negativo**

prestazioni della ricerca sequenziale

ecco una codifica di questo algoritmo come funzione C++, dove i parametri **l**, **r** sono indici che delimitano lo spazio di ricerca del valore **v** nell'array **a**, dunque **N = r-l+1**, e si ha il valore di ritorno -1 quando la ricerca ha esito negativo:

```
int search(int a[], int v, int l, int r)
  {
    for (int i = l; i <= r; i++)
      if (v == a[i]) return i;
    return -1;
  }
```

la ricerca sequenziale esamina mediamente

N elementi in caso di esito negativo

all'incirca **N/2** elementi in caso di esito positivo

ordinando preventivamente l'array (costo $\propto N \lg N$, irrisorio quando **M >> N** perché sostenuto una tantum), si può dimezzare il costo medio della ricerca sequenziale ad esito negativo, arrestandola quando il valore dell'elemento dell'array è maggiore di quello cercato (se l'array è ordinato per valori crescenti)

il costo della ricerca sequenziale è sempre **N** nel caso peggiore, in qualsiasi esito

ricerca binaria

l'ordinamento preventivo dell'array, che produce un modesto guadagno prestazionale nella ricerca sequenziale, è sfruttato molto meglio nell'algoritmo di **ricerca binaria**, che ad ogni passo con esito negativo dimezza la dimensione dello spazio di ricerca nel passo successivo

ecco una codifica dell'algoritmo come funzione C++:

```
int search(int a[], int v, int l, int r)
  {
    while (r >= l)
      { int m = (l+r)/2;
        if (v == a[m]) return m;
        if (v < a[m]) r = m-1; else l=m+1;
      }
    return -1;
  }
```

dallo studio già effettuato della ricorrenza $C_N = C_{N/2} + 1$ possiamo concludere che il costo della ricerca binaria è proporzionale a $\lg N$ (anche nel caso peggiore)

il **guadagno di efficienza** rispetto alla ricerca sequenziale è **esponenziale!**

complessità di algoritmi e di problemi

finora si sono considerate stime e analisi delle prestazioni di **algoritmi** per un dato problema, o classe di problemi, e per essi risulta utile distinguere fra

caso peggiore : l'analisi delle prestazioni in tal caso fornisce **garanzie** sul costo computazionale dell'algoritmo

caso medio : l'analisi delle prestazioni in tal caso fornisce una **stima** del costo computazionale mediamente atteso

per complessità computazionale di un **problema** si intende:

il costo computazionale nel caso peggiore del miglior algoritmo per quel problema

il costo computazionale nel caso peggiore di un algoritmo per un dato problema costituisce un **limite superiore** alla complessità computazionale del problema:

il costo del miglior algoritmo non può superare quello di qualsiasi algoritmo

è anche utile determinare **limiti inferiori** alla complessità computazionale di problemi, analizzandone caratteristiche inerenti, ma questo compito è di solito più difficile

esercizi

1. mostrare che la soluzione della ricorrenza $C_N = C_{N/2} + 1$ è $\approx \lg N$ anche con l'approssimazione intera per eccesso del dimezzamento: $N/2 = \lceil N/2 \rceil$
2. mostrare che la soluzione della ricorrenza $C_N = C_{N/2} + N$ è $\approx 2N$ anche con l'approssimazione intera per eccesso del quoziente: $N/2^k = \lceil N/2^k \rceil$
3. risolvere la ricorrenza $C_N = kC_{N/2}$ per $N > 1$, con $C_1 = 1$, per il caso in cui $N = 2^n$
4. modificare la codifica C++ della funzione di ricerca sequenziale, assumendo che l'array **a** sia ordinato per valori crescenti, in modo che la ricerca termini con esito negativo quando il valore dell'elemento dell'array è maggiore di quello cercato