Programmazione Orientata Agli Oggetti

Concetti di complessità degli algoritmi

Libero Nigro

La complessità (o efficienza) di un algoritmo può essere *complessità temporale,* che mira a caratterizzare il tempo di esecuzione dell'algoritmo al variare della dimensione dei dati, o *complessità spaziale,* che valuta l'ingombro di memoria richiesto dai dati dell'algoritmo, sempre al variare della numerosità dei dati.
In quanto segue, l'attenzione è posta sulla complessità temporale.

Piuttosto che tentare di "misurare" il tempo di esecuzione richiesto da un algoritmo su una certa piattaforma (la misura, infatti, è sensibile al tipo di compilatore adoperato, al sistema operativo sottostante, all'hardware della macchina etc.), l'obiettivo è estrarre il *comportamento intrinseco* dell'algoritmo al variare della dimensione dell'input.
Ad esempio, volendo caratterizzare la complessità di un algoritmo di ordinamento di un array con **n** elementi, di interesse non è tanto conoscere "esattamente" il tempo impiegato dall'algoritmo, quanto la dipendenza del comportamento temporale dell'algoritmo al crescere di n (*comportamento asintotico*). Per chiarire le idee, di seguito si considera l'algoritmo di selection sort su un array di interi **a** di n elementi (n potrebbe essere a.length).

Complessità "esatta" di Selection Sort

```
for( int j=n-1; j>0; j--){
   int iMax=0;
   for( int i=1; i<=j; i++ ) //for-interno
       if( a[i]>a[iMax] ) iMax=I;
   int tmp=a[j];
   a[j]=a[iMax];
   a[iMax]=tmp;
}//for-esterno
```

Si definisce "complessità esatta" di un algoritmo il conteggio delle operazioni eseguite per realizzare l'obiettivo, in questo caso portare a termine l'ordinamento. A questo proposito si assume, per semplicità, che tutte le operazioni elementari (un confronto, un'assegnazione etc.) abbiamo un costo temporale unitario.
Per la funzione complessità esatta di selection sort, che dipende dalla dimensione dell'input n, si ha:

$$T_{SS}(n) = 2 + num - operazioni - ciclo\ esterno$$

Il 2 si riferisce alla inizializzazione del for esterno e all'ultimo test quando j==0 e si esce dal for. Contando le operazioni eseguite dal for-esterno si ha:

$$T_{SS}(n) = 2 + 8(n-1) + operazioni - for - interno$$

Ad ogni girata del ciclo esterno, si eseguono sempre 8 operazioni: l'inizializzazione di iMax, l'inizializzazione del for interno, la condizione di uscita del for interno (quando i>j), le tre assegnazioni per lo scambio di a[iMax] con a[j], la condizione del for esterno, il passo del for esterno. Siccome il ciclo esterno è ripetuto (n-1) si ha il primo contributo 8*(n-1). Manca ancora il computo delle operazioni eseguite dal for interno.
Quando j vale (n-1), il ciclo interno è ripetuto (n-1) volte, quando j vale (n-2), il for interno è ripetuto (n-2) volte etc. Considerando che ad ogni iterazione del for interno, nel *caso peggiore* (in generale l'algoritmo dovrebbe essere studiato nei tre casi: peggiore, migliore e medio dei dati) si compiono 4 operazioni (verifica condizione di continuazione, confronto ed assegnazione di un nuovo valore ad iMax, ed il passo del for (i++)), ne deriva che la sommatoria delle varie iterazioni del ciclo interno vanno moltiplicate per 4.
Ricordando l'equivalenza di Gauss sulla somma dei primi n numeri naturali (1+2+3+…+(n-1)+n):

$$\sum_{i=1}^{n} i = \frac{n*(n+1)}{2}$$

si riconosce subito che entro le parentesi quadre c'è la somma dei primi (n-1) naturali, dunque:

$$\sum_{i=1}^{n-1} i = \frac{(n-1)*n}{2}$$

Pertanto:

$$T_{SS}(n) = 2 + 8(n-1) + 2(n-1)n = 2n^2 + 6n - 6$$

Big O (ordine di) e analisi asintotica per n→∞

Si dice che una funzione T(n) è dell'ordine di una funzione f(n) e si scrive $T(n) = O(f(n))$ se si possono trovare due costanti positive $a \; e \; n_0$ tali che: $T(n) \leq a * f(n) \; \forall n \geq n_0$

Nel caso di selection sort si ha subito: $T_{SS}(n) = O(n^2)$

Infatti scegliendo ad es. a=3, già per n_0=1 si ha: $T_{SS}(n) \leq 3n^2 \; \forall \; n \geq 1$

Si nota che i valori delle costanti a ed n_0 potrebbero essere scelti più grandi e la relazione $T(n) \leq a * f(n) \; \forall n \geq n_0$ varrebbe a maggior ragione. Si nota ancora che, in base alla definizione dell'operatore O, risulta pure:

$$T_{SS}(n) = O(n^3) \;\; ed \; anche \;\; T_{SS}(n) = O(n^k), \qquad k \geq 2$$

I risultati che precedono dicono che per stimare l'ordine di grandezza (O) del tempo di esecuzione di un algoritmo è sufficiente ignorare i coefficienti e termini di grado inferiore nell'espressione della complessità esatta. Inoltre, dire che $T_{SS}(n) = O(n^2)$ significa dire che il tempo vero di esecuzione dell'algoritmo su qualsiasi calcolatore, cresce col quadrato della dimensione dell'input, ossia è proporzionale ad n^2 a meno di qualche costante.
Si osserva, infine, che $T(n) = O(f(n))$ significa che la funzione T(n) è *dominata* dalla f(n) non appena la dimensione dell'input supera una certa soglia minima n_0.

Operatore Omega grande (Ω)

Si dice che una funzione T(n) = Ω(f(n)) se si possono trovare due costanti positive a ed n_0 tale che: $T(n) \geq a * f(n) \; \forall n \geq n_0$
ossia la funzione T(n) *domina* la f(n) appena n supera una soglia n_0.

Operatore Teta grande (Θ)

Si dice che una funzione T(n) = Θ(f(n)) se si possono trovare tre costanti positive a1, a2 ed n_0 tale che: $a_1 * f(n) \leq T(n) \leq a_2 * f(n) \; \forall n \geq n_0$

In questo caso la T(n) cresce *come* la f(n) non appena n supera la soglia n_0.

Si verifica facilmente che per n>1, $2n^2+6n-6$ è compresa tra n^2 e $3n^2$.

All'atto pratico si preferisce utilizzare la notazione Big O e scrivere O(f(n))
ma col significato dell'operatore Θ grande, ossia con riferimento alla "più piccola"
funzione dominante f(n).

- **Alcune complessità**
- Ricerca in una tabella hash: $T_{hash}(n) = O(1)$ ossia l'operazione richiede un tempo (quasi) costante.
- Ricerca lineare: $T_{RL}(n) = O(n)$
- Ricerca binaria: $T_{RB} = O(\log_2 n) = O(\log n)$ sotto intendendo la base 2 per i logaritmi
- Tutti i metodi elementari di ordinamento: $T_{OE}(n) = O(n^2)$ in cui OE può essere selection sort, bubble sort, insertion sort, etc.

Metodi avanzati di ordinamento: $T_{OA}(n) = O(n \log n)$ dove OA può essere merge sort, heap sort, quick sort etc.

Per il problema dell'ordinamento, basato su confronti e scambi, sussiste il risultato: T(n)= Ω(n log n) ossia nessun algoritmo esiste con una complessità inferiore a O(n*log n).

Si dicono *algoritmi polinomiali*, tutti gli (usuali) algoritmi che hanno complessità del tipo $O(n^k)$ per qualche k. Gran parte dei problemi sono (per nostra fortuna) risolvibili in tempi polinomiali.

Tuttavia esistono algoritmi con *complessità esponenziale*, es. $T(n)=O(2^n)$.
Tali algoritmi ed i relativi problemi sono spesso detti intrattabili, nel senso che all'atto pratico, per n non troppo piccolo (es. 50), l'algoritmo non termina, quale che sia il calcolatore utilizzato. In questi casi si ricorre a *metodi euristici* per ottenere soluzioni approssimate del problema.

Un'altra complessità rilevante è la complessità fattoriale: O(n!). Al crescere di n, il tempo di calcolo cresce in modo notevole.

Il metodo bucket sort

Si vuole ordinare una successione di numeri che, si sa, sono compresi nell'intervallo 0-100. L'ordinamento si può banalmente ottenere come segue:

```
public class BucketSort{
    public static void main( String[] args ){
        Scanner sc=new Scanner(System.in);
        int []b=new int[101]; //indici da 0 a 100
        //l'array b è inizializzato a tutti 0
        for(;;){
            int x=sc.nextInt();
            if( x<0||x>100) break;
            b[x]++; //conta questo x
        }
        System.out.print("[");
        for( int i=0; i<b.length; ++i )
            if( b[i]>0 )
            for( j=0; <b[i]; ++j )
                System.out.print(i+" ");
        System.out.println("]");
    }//main
}//BucketSort
```

Gli elementi di b sono bucket e funzionano come contatori del relativo indice.

Una volta riempito b, è sufficiente scrivere b[i] volte il valore i per tutte le i per avere l'ordinamento!

L'algoritmo è chiaramente O(n) ma non sconfessa i risultati generali dal momento che **non** si basa su confronti e scambi.

Un algoritmo esponenziale: il bin-packing

- Si tratta di riempire contenitori standard con n oggetti in modo da utilizzare il minor numero di scatoli
- Consideriamo contenitori standard di volume 1.0, e 4 prodotti di volume 0.6, 0.4, 0.2, 0.5. Quanti scatoli occorrono al min?
- Siccome il problema ha complessità esponenziale $O(2^n)$ si usa spesso l'euristica di ordinare preliminarmente gli oggetti, e riempire in successione gli scatoli procedendo secondo l'ordine.
- Si ha: 0.2, 0.4, 0.5, 0.6 e occorrono tre scatoli: <0.2,0.4>,<0.5>,<0.6>. Si tratta di una soluzione sub-ottimale, dal momento che una soluzione ottima può basarsi su 2 scatoli, es.

  <0.4,0,6>,<0.2,0.5> o

  <0.6,0.2>,<0.4,0.5>

Per la soluzione ottima si devono ricercare tutti i possibili raggruppamenti o sottinsiemi di oggetti e con questi riempire gli scatoli.