Lower Bound per Algoritmi di Ordinamento

Davide Aversa

24 novembre 2009

Indice

1 Introduzione **3**
1.1 Il concetto di Lower–Bound 3
1.2 Calcolare il Lower–Bound 3
1.3 L'algoritmo ottimo 4

2 Dimostrazione **5**
2.1 L'albero decisionale 5
2.2 Profondità di un albero decisionale 7

1 Introduzione

1.1 Il concetto di Lower–Bound

Nello studio della complessità di un algoritmo esistono molti valori rilevanti il ruolo principe è occupato dalla ricerca dell' *Lower–Bound* di un algoritmo. Il lower–bound rappresenta infatti una sorta di "barriera matematica invalicabile" alla *semplicità* di un algoritmo: esso è il minimo valore possibile che può assumere la complessità di un *qualunque* algoritmo che risolva il problema X.

Definizione 1 Detto in modo più rigoroso, supponiamo di avere un problema X e definiamo l'insieme F come l'insieme di tutti gli algoritmi che risolvono il problema X. Inoltre definiamo la funzione $C(f)$ che valuta la complessità dell'algoritmo $f \in F$. A questo punto il lower–bound del problema X sarà definito da:

$$\inf\{s : s = C(f) \wedge f \in F\} \tag{1}$$

ovvero come la complessità dell'algoritmo più "veloce" fra tutti gli algoritmi possibili.

Da qui possiamo subito capire che non può esistere un algoritmo che abbia una complessità di ordine inferiore al lower bound. La dimostrazione è semplice: se supponiamo che esista un algoritmo $\overline{f}$ che risolve il problema X tale che la sua complessità sia minore del *lower–bound* di X allora avremo che $\overline{f} \notin F$ e quindi, per definizione, non risolve il problema X, con ovvia contraddizione.

1.2 Calcolare il Lower–Bound

Abbiamo una definizione di lower–bound ma non abbiamo un metodo per calcolarlo. La difficolta giunge dal fatto che per essere sicuri che un certo valore sia il lower-bound del problema dobbiamo calcolare la complessità di *tutti gli algoritmi possibili che risolvono X*. Un compito ovviamente improponibile: come possiamo essere sicuri che non esista un algoritmo più efficente dell'algoritmo più efficente attualmente conosciuto?

In alcuni casi però è possibile, tramite alcuni ragionamenti matematici, modellizzare un algoritmo "teorico" e del tutto generico che raccoglie l'essenza di un problema. Calcolando la complessità nel caso peggiore per questo algoritmo possiamo, con certezza, stabilire che non esiste un algoritmo con una complessità minore di quella del nostro "algoritmo virtuale".

Questo è proprio il caso per gli algoritmi di ordinamento.

1.3 L'algoritmo ottimo

La conoscenza del *lower–bound* ci permette anche di definire l'*algoritmo ottimo*. Un algoritmo ottimo non è altro che un algoritmo con complessità equivalente al lower-bound.

Per definizione non può esistere un algoritmo più efficente.

2 Dimostrazione

2.1 L'albero decisionale

Per dimostrare il lower–bound degli algoritmi di ordinamento dobbiamo innanzitutto conoscerne l'essenza: un algoritmo di ordinamento non è altro che un algoritmo in grado di distinguere, fra tutte le permutazioni di un insieme, la permutazione che rispetta i principi di ordinamento richiesti. Ad esempio supponiamo, per semplicità, di dover ordinare un insieme di tre valori: `a`, `b` e `c`. Nel nostro caso semplificato avremo le seguenti permutazioni:

- `a-b-c`
- `a-c-b`
- `b-a-c`
- `b-c-a`
- `c-a-b`
- `c-b-a`

Nel caso di più generale avremo un numero di permutazioni pari a $n!$ dove n è il numero di elementi da ordinare. Ovviamente non è necessario provare tutte le permutazioni perché possiamo notare facilmente che, facendo un semplice confronto, riusciamo ad escludere in un solo colpo un certo numero di permutazioni.

Ad esempio, se appuriamo con un confronto che `a<b`, possiamo escludere automaticamente le permutazioni `b-a-c`, `b-c-a` e `c-b-a` (ho supposto che le permutazioni rappresentino liste in ordine crescente). A questo punto possiamo effettuare un nuovo confronto, ad esempio `b<c`, grazie al quale possiamo escludere anche le restanti permutazioni lasciando l'unica lista possibile: `a-b-c`.

Quello che, in pratica, viene fatto durante l'ordinamento non è altro che effettuare dei confronti fra gli elementi ad ogni passo ed eliminare man mano gruppi di permutazioni in base all'esito.

Questo si può rappresentare grazie ad un *albero decisionale* come quello mostrato in figura 1. Un albero decisionale è un albero binario che ha come nodi interni insiemi di permutazioni e come nodi foglia una sola permutazione. Ogni livello dell'albero corrisponde ad un confronto grazie al quale si può *decidere* se andare nel sotto-albero sinistro o nel sotto albero destro. Il nodo radice conterrà l'insieme N_0 contenente tutte le $n!$ permutazioni degli elementi della lista. Passando al prossimo livello verrà effettuato un confronto qualsiasi (ad esempio $n < m$ dove sia n che m sono elementi della lista) e, se verificato scenderemo a sinistra, altrimenti a destra. Il nodo

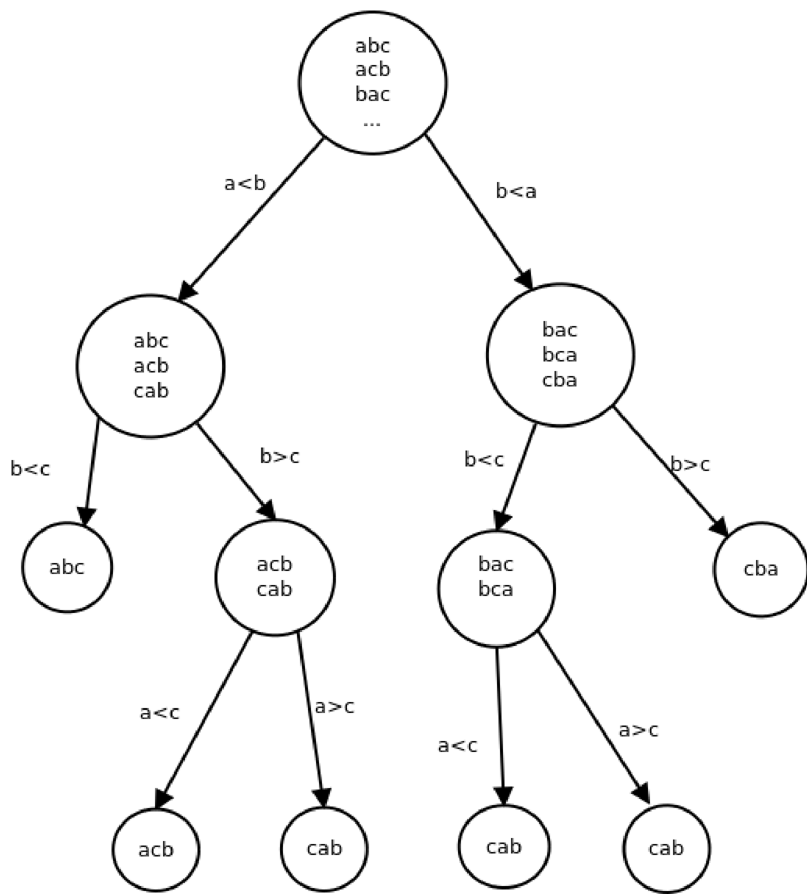

Figura 1: Albero decisionale dell'insieme a-b-c.

di destinazione conterrà quindi solo le permutazioni *coerenti con l'esito del confronto.*

Un altra proprietà dell'albero decisionale sopra definito consiste nel seguente lemma:

Lemma 1

$$N = N_s \oplus N_d \tag{2}$$

Ovvero che l'insieme contenuto in ogni nodo è *somma diretta* degli insiemi contenuti nei due suoi figli. La dimostrazione di questo fatto è banale: poiché abbiamo già visto che ogni nodo figlio contiene solamente le permutazioni coerenti con l'esito di un confronto e quindi non può esistere una permutazione che appartenga sia a N_s che a N_d poiché, in quel caso, significherebbe che per un certo elemento n sia vero sia che $n < m$ sia che $m \leq n$.

2.2 Profondità di un albero decisionale

A questo punto ci resta facile capire qual è il numero massimo di decisioni che dobbiamo fare. La complessità del problema corrisponde infatti al numero di confronti che dobbiamo attraversare per arrivare ad un nodo foglia.

Sappiamo che il numero di nodi foglia è esattamente $n!$. A questo punto, poiché l'albero di decisione è un albero binario, possiamo calcolare con facilità l'altezza dell'albero e, quindi, la complessità dell'algoritmo.

Definizione 2 La complessità minima di un algoritmo di ordinamento è:

$$C_{sort} = o(n \log n) \tag{3}$$

Dimostrazione Poiché l'albero in questione è binario possiamo calcolare il numero di nodi dell'albero in base alla sua altezza. Definendo $q(h)$ come la funzione che restituisce il numero di nodi foglia di un albero binario di altezza h possiamo dimostrare per induzione che:

Passo Base

$$q(1) \leq 2 \tag{4}$$

Ipotesi Induttiva Supponiamo che sia vero che

$$q(h) \leq 2^h \tag{5}$$

Infatti $q(1) \leq 2^1 = 2$

Passo Induttivo Dimostriamo che $q(h+1) \leq 2^{h+1}$.

Un albero di altezza $h+1$ può essere rappresentato come un nodo con due sotto-alberi di altezza h. E quindi il numero di foglie sarà dato dalla somma delle foglie dei sotto-alberi. Perciò possiamo definire la seguente serie di disequazioni:

$$q(h+1) = q(h) + q(h) = 2q(h) \leq 2 * (2^h) = 2^{h+1} \tag{6}$$

A questo punto possiamo quindi definire come $\overline{q}(h)$ come:

$$\overline{q}(h) = \max\{q(h)\} = 2^h \tag{7}$$

Ovvero il massimo numero di foglie di un albero binario di altezza h e possiamo quindi invertire la funzione ricavando una funzione che ci dia l'altezza in base al numero di foglie.

$$h = \log q \tag{8}$$

Dove il logaritmo è da intendersi in base 2. Tornando al problema dell'ordinamento possiamo sostituire a q il numero di foglie dell'albero decisionale, ovvero $n!$

$$h = \log n! \tag{9}$$

Da cui segue, in virtù dell'*approssimazione di Stirling*, che:

$$h \geq n \log n \tag{10}$$

da cui deriva che la complessità del problema è:

$$C_{sort} = o(h) = o(\log n!) = o(\log n^n) = o(n \log n) \tag{11}$$

Da cui la tesi.