Brevi note sulla complessità computazionale

Versione del 2 Aprile 2020

Comparazione di algoritmi

Spesso vorremmo poter confrontare algoritmi differenti che risolvono lo stesso problema, come ad esempio l'ordinamento di un array di valori. Supponiamo che questi algoritmi siano chiamati A e B. Dato che sia A che B risolvono lo stesso problema, dobbiamo compararli in termini di risorse utilizzate, in particolare il tempo che ci mettono ad eseguire (complessità *temporale*) e la quantità di memoria che occupano (complessità *spaziale*). Noi ci occuperemo principalmente della complessità temporale di un algoritmo, il che significa che vogliamo quantificare la nozione intuitiva di "va più veloce".

Un modo empirico di stabilire che algoritmo va più veloce tra A e B è quello di eseguire i due algoritmi e verificare i tempi di esecuzione. Supponiamo di voler ordinare un array di 1000 elementi e di ottenere i seguenti tempi:

Algoritmo	Tempo
A	2s
B	5s

Possiamo quindi concludere che A è più veloce di B in generale? No, perché questo è il risultato su una specifica macchina e per una specifica dimensione dell'input (1000 elementi). Magari su una macchina diversa con 10000 elementi i risultati ottenuti sarebbero:

Algoritmo	Tempo
A	200s
B	30s

La situazione sembra essersi ribaltata! A meno di non conoscere esattamente la macchina e la dimensione attesa dell'input non siamo facilmente in grado di stabilire quale algoritmo scegliere. Per questo usiamo un'astrazione e supponiamo che tutta una serie di istruzioni elementari (assegnamento, operazioni aritmetiche, comparazioni, etc) richiedano un tempo costante e vogliamo esprimere il costo in termini di tempo di un algoritmo come una funzione che ci dica il numero di passi elementari che l'algoritmo esegue in funzione della dimensione dell'input (e.g., numero di elementi).

Algoritmo	Numero di passi in funzione della dimensione dell'input n
A	$2n^2 + 3n + 4$
B	$n + 100$

Avendo queste funzioni possiamo sapere quanti passi elementari svolge ognuna di esse a seconda dell'input e possiamo quindi scegliere quale algoritmo utilizzare. Data la complessità dei computer moderni, il conto dei passi elementari non ci dice tutto (non tutte le operazioni sono eseguite nello stesso tempo, l'accesso alla memoria non è uniforme, etc) ma ci fornisce una prima approssimazione da utilizzare.

Caso peggiore e caso migliore

A volte il numero di passi elementari compiuti da un algoritmo non dipende solo dalla dimensione dell'input, ma dall'input specifico. Quindi, per fare un esempio, con un particolare array di n elementi un algoritmo può richiedere poche operazioni, mentre con un altro input sempre di dimensione n potrebbe richiederne molte di più.

Per questo distinguiamo l'analisi del *caso migliore* da quella del *caso peggiore*. Possiamo avere una funzione che ci dice il numero di passi elementari effettuati nel caso migliore (ovvero, prendendo il minimo dei passi tra tutti i possibili input di lunghezza n) ed un'altra funzione che ci dice il numero di passi elementari effettuati nel caso peggiore (prendendo il massimo dei passi tra tutti i possibili input di lunghezza n).

Potremmo ad esempio avere due algoritmi A' e B' che hanno i seguenti casi migliori e peggiori:

Algoritmo	Caso migliore	Caso peggiore
A'	30	$n^3 + 8n + 35$
B'	$n + 50$	$2n + 67$

Notiamo come ci possa essere una ampia differenza tra caso migliore e peggiore quindi, a volte, è utile parlare di caso medio, in cui prendiamo il numero medio di passi elementari effettuati su tutti gli input di lunghezza n.

Quella che usiamo più spesso (quasi sempre) è l'analisi del caso peggiore, perché rappresenta – per l'appunto – il caso peggiore possibile e, con un certo pessimismo, vogliamo essere sicuri di avere un limite a "quando male" il nostro algoritmo andrà.

Notazione asintotica

Fino ad adesso abbiamo trattato il caso migliore e peggiore del numero di passi elementari (diremo, più semplicemente, del tempo richiesto) dando funzioni in grado di restituirci il numero esatto di passi che sono richiesti. Però potrebbe non essere necessario andare così nel dettaglio. Supponiamo ad esempio di voler comparare un algoritmo che richiede $37n^2 + n + 4$ passi nel caso peggiore con uno che ne richiede $8n^3 + 2n^2 + 6$. Ci interessano realmente i termini di grado inferiore? Generalmente no. Sappiamo che una funzione cubica supererà sempre in valore una funzione quadratica per n abbastanza grande, quindi a noi basterebbe sapere asintoticamente chi cresce di più tra le diverse funzioni. Per questo definiamo tre diverse classi di funzioni:

- **O grande**. Data una funzione $g(n)$, la classe di $O(g(n))$ è la classe di tutte le funzioni $f(n)$ tali per cui esistono due costanti $c > 0$ e $n_0 > 0$ tale per cui per per ogni $n > n_0$ abbiamo che $f(n) \leq cg(n)$. In pratica stiamo dicendo che $f(n)$ è, per n abbastanza grande, limitata superiormente da $cg(n)$. Con un comune abuso di notazione diremo che $f(n) = O(g(n))$ per indicare che $f(n) \in O(g(n))$. Possiamo ad esempio dire che il costo in termini di tempo di un algoritmo nel caso peggiore è $O(n^2)$ per indicare che non può fare peggio di una funzione quadratica. Magari fa anche molto meglio (per esempio richiede solo un tempo che cresce linearmente) ma sicuramente non fa peggio.
- Ω **grande**. Data una funzione $g(n)$, la classe di $O(g(n))$ è la classe di tutte le funzioni $f(n)$ tali per cui esistono due costanti $c > 0$ e $n_0 > 0$ tale per cui per per ogni $n > n_0$ abbiamo che $cg(n) \leq f(n)$. Questo è l'opposto di O grande, stiamo dicendo che $f(n) = \Omega(g(n))$ se la funzione $f(n)$ cresce asintoticamente almeno quanto $cg(n)$. Questo significa che se abbiamo un algoritmo che nel caso peggiore richiede tempo $\Omega(n \log n)$, significa che non possiamo fare meglio di $cn \log n$. Magari facciamo molto peggio (e.g., una funzione quadratica) ma sicuramente non possiamo scendere ad un tempo che è lineare in n.
- Θ. Data una funzione $g(n)$, la classe di $O(g(n))$ è la classe di tutte le funzioni $f(n)$ tali per cui esistono tre costanti $c_1, c_2 > 0$ e $n_0 > 0$ tale per cui per per ogni $n > n_0$ abbiamo che $c_1 g(n) \leq f(n) \leq c_2 g(n)$. Questo significa che $f(n)$ "cresce quanto" $g(n)$. Se un algoritmo richiede nel caso peggiore un tempo $\Theta(n)$, questo significa che ci mettiamo esattamente un tempo lineare rispetto a n. Rimangono nascoste le costanti c_1 e c_2, ma sappiamo quanto $f(n)$ cresce.

Se vogliamo riassumere in modo un poco impreciso il compito di queste notazioni, $O(g(n))$ ci dice "al peggio ci mettiamo un numero di passi che è proporzionale a $g(n)$", $\Omega(g(n))$ ci dice "non facciamo mai meglio di un numero di passi proporzionale a $g(n)$", e $\Theta(g(n))$ ci dice "Il numero di passi richiesti cresce – a meno di costanti moltiplicative – g(n)".

Un esempio pratico

Supponiamo di voler calcolare il costo in termini di tempo (la complessità temporale) della seguente funzione in Python in funzione della lunghezza n dell'array `A`:

```
def func(A):
    somma = 0
    for i in range(0, len(A)):
        somma = somma + A[i]
    return somma
```

analizziamo riga per riga il costo delle operazioni:

```
    somma = 0
```

questo è un assegnamento che costa 1, una singola istruzione elementare. La riga successiva

```
    for i in range(0, len(A)):
```

viene eseguita un numero di volte pari alla lunghezza di `A`, quindi n volte così come sarà eseguita n volte ogni istruzione all'interno del ciclo for, come è

```
        somma = somma + A[i]
```

che, sebbene costi una sola operazione (o, a seconda delle assunzioni, anche due avendo sia una somma che un assegnamento), viene eseguita ogni volta che il ciclo for viene eseguito. Infine, l'operazione

```
    return somma
```

viene eseguita una volta sola e ha costo unitario.

Ne segue che il costo dell'algoritmo è, in funzione di n, $T(n) = 2n+2$ (indichiamo con $T(n)$ il tempo richiesto). Sfruttando le definizioni precedenti, si può vedere come $T(n)$ sia limitato sia inferiormente che superiormente da funzioni lineari (per esempio n come limitazione inferiore e $3n$ come limitazione superiore), quindi diremo che $T(n) = \Theta(n)$.