Cenni sulla complessità computazionale

Alessandro Bugatti

"In teoria non c'è differenza fra teoria e pratica.

In pratica la differenza c'è"

Introduzione

In questa dispensa si daranno dei cenni della teoria della complessità computazionale come strumento per misurare la "bontà" degli algoritmi che risolvono un dato problema, non soffermandosi sulla trattazione matematica, ma cercando di offrire dei concetti che permettano di apprezzare almeno da un punto di vista qualitativo il problema nel suo complesso.

Algoritmi e programmi

Come già appreso in precedenza un algoritmo consiste in una serie ben definita di passi che portano alla risoluzione di un dato problema. Pur non essendoci una definizione universalmente condivisa possiamo dire che un algoritmo deve soddisfare queste 4 caratteristiche:

- il numero di passi che lo compone deve essere finito
- deve terminare con un risultato che è la soluzione del problema
- le istruzioni devono essere elementari e immediatamente eseguibili
- le istruzioni devono essere espresse chiaramente, in modo che la loro interpretazione sia univoca

Quindi dato un problema il primo scopo è quello di trovare l'algoritmo risolutore. Spesso però un problema può essere risolto in diversi modi tramite algoritmi diversi ma tutti corretti. La questione che affronteremo sarà dunque quella di riuscire a caratterizzare un algoritmo corretto in modo tale da poterlo confrontare con un altro algoritmo anch'esso corretto e poter decidere quale sia "migliore" per risolvere lo stesso problema.

Memoria e velocità di esecuzione

Per stabilire quale sia l'algoritmo migliore per risolvere un determinato problema dobbiamo prima definire rispetto a che caratteristiche possiamo dire che un algoritmo è migliore di un altro. Nella realtà quotidiana dei programmatori ci potrebbero essere molti parametri su cui misurare la qualità di un algoritmo e quindi sceglierlo: la possibilità di averlo già pronto all'uso perchè presente in una libreria, la facilità di implementazione, la possibilità di adattarsi bene alla particolare architettura utilizzata, la semplicità di comprensione ecc.

In questa dispensa però vogliamo analizzare delle caratteristiche fondamentali inerenti l'algoritmo e non aspetti esterni legati al suo essere un programma per computer. Solitamente i due aspetti più strettamente legati ad un algoritmo sono l'utilizzo di memoria e la velocità di esecuzione. L'utilizzo della memoria non verrà trattato, in quanto rimane un aspetto fondamentale solo in quei contesti in cui la memoria stessa è una risorsa scarsa (vedi ad esempio alcuni dispositivi *embedded*). Quello che invece ci interessa maggiormente è la velocità di esecuzione e come prima cosa bisogna trovare un modo per definirla in maniera chiara.

Un algoritmo non è legato solo al mondo dei computer (si pensi ad esempio ad una ricetta di cucina che se sufficientemente dettagliata può essere vista come un algoritmo), ma quando viene implementato su un computer bisogna definire i passaggi che attraversa per sapere che tipo di velocità intendiamo misurare. Nel mondo dei computer il primo passaggio che subisce un algoritmo, dopo la sua ideazione, è quello di essere trasformato

in un diagramma di flusso o in uno pseudo-codice. Questa è la forma più pura di un algoritmo, in quanto esso viene descritto in astratto, ma in questa forma non può essere utilizzato sul calcolatore e necessita quindi di ulteriori passaggi. Solitamente a questo punto l'algoritmo viene tradotto in un linguaggio di programmazione e, a seconda del tipo di linguaggio, viene o reso eseguibile (se si usa un linguaggio compilato) oppure interpretato (se si usa un linguaggio interpretato). A questo punto il programma viene eseguito e diventa un processo in memoria. Si può quindi vedere che agli estremi di questa catena si trovano l'algoritmo in forma simbolica e il processo in esecuzione.

Potrebbe sembrare semplice confrontare tra loro due algoritmi misurando "con un orologio" il tempo di esecuzione di entrambi sullo stesso insieme di input: per quanto precedentemente detto però questo confronto, per avere un qualche significato, dovrebbe essere fatto in condizioni molto controllate, poichè nel caso di un processo il tempo di esecuzione può essere influenzato da diversi fattori quali:

- il linguaggio con cui è stato programmato: in generale i linguaggi compilati producono programmi più veloci dei linguaggi interpretati e ogni linguaggio può avere delle caratteristiche che lo rendono più veloce in determinate aree (ad esempio un programma scritto in C++ è generalmente più veloce dello stesso programma scritto in Java)
- la bontà del compilatore o dell'interprete: un codice scritto con lo stesso linguaggio di programmazione può essere sottoposto a compilatori diversi che, pur producendo programmi funzionalmente equivalenti, possono essere formati da diverse sequenze di codici macchina e anche questo può ripercuotersi sulla velocità di esecuzione
- l'abilità di chi ha scritto il programma: lo stesso algoritmo può essere scritto in modi diversi e questo può riflettersi direttamente sull'efficienza del programma
- l'ambiente in cui gira il programma: il sistema operativo su cui il programma viene fatto girare impatta sull'efficienza di alcune istruzione (tipicamente gestione delle memoria, I/O, threading, ecc.) influendo sul tempo totale di esecuzione
- la macchina sulla quale gira il programma: è ovvio che i componenti hardware di un PC influenzano la velocità di esecuzione delle istruzioni, principalmente la frequenza della CPU, ma non solo

Sembra quindi evidente che la misura "con l'orologio" può avere un qualche significato solo se gli algoritmi da confrontare sono scritti nello stesso linguaggio dalla stessa persona, compilati con lo stesso compilatore ed eseguiti sulla stessa macchina (in realtà anche in questo caso potrebbero esserci delle differenze che possono falsare il confronto).

L'algoritmo in forma simbolica ha invece il vantaggio di essere una rappresentazione astratta, indipendente dai fattori visti sopra. La domanda quindi è: come facciamo a "misurare" la velocità di un algoritmo astratto senza usare l'orologio? Intuitivamente possiamo pensare di contare il numero di istruzioni che occorrono per eseguire un determinato compito: più istruzioni equivalgono ad un algoritmo più lento, meno istruzioni ad un algoritmo più veloce. Per comprendere meglio quanto detto partiamo da un semplice esempio, la moltiplicazione tra due numeri. Supponiamo inoltre che la moltiplicazione non sia un'istruzione semplice ma debba essere eseguita come una serie di somme, per cui ad esempio 2×3 risulta $2 + 2 + 2$. Se l'operazione che dobbiamo eseguire è 3×113 l'algoritmo più immediato è quello di eseguire un ciclo per 113 volte e ogni volta sommare il numero 3 al risultato ottenuto al giro precedente. Questo comporta l'esecuzione di 113 somme, 113 confronti (ad ogni giro bisogna controllare se l'algoritmo può terminare o deve andare avanti) e 113 incrementi (la variabile del ciclo). Se approssimativamente consideriamo uguali i costi di queste operazioni otteniamo che in questo caso specifico l'algoritmo deve eseguire $113 + 113 + 113 = 339$ istruzioni.

Pensandoci meglio potremmo trovare un algoritmo migliore anche per questo semplice problema: se prima di iniziare il ciclo controllassimo quale dei due operandi è il minore potremmo in alcuni casi ridurre drasticamente il costo dell'algoritmo. Nell'esempio potremmo scambiare il 3 con il 113 e effettuare solo 3 giri nel ciclo, quindi il costo risulterebbe quello del controllo iniziale, dello scambio delle variabili che prevede 3 assegnamenti e la somma dei costi per i 3 giri e quindi $1 + 3 + 3 + 3 + 3 = 13$ istruzioni.

Passando alla generalizzazione dell'esempio otteniamo che il problema da risolvere è quello di vedere quante istruzioni sono necessarie per eseguire un prodotto del tipo $N \times M$: nel caso del primo algoritmo il costo sarà di $3 \times M$ istruzioni, mentre nel secondo caso avremo $1 + 3 + 3 \times \min(N, M)$ (dove il 3 in realtà non è sempre

presente). Come si può intuitivamente capire quando $M \gg N$ il secondo algoritmo è decisamente migliore del primo (come nell'esempio mostrato), mentre nei casi in cui il primo è migliore lo è solo di poco.

Riassumendo possiamo dire che per misurare la velocità di un algoritmo dobbiamo contare il numero di istruzioni che esegue rispetto alla dimensione del suo input (nell'esempio precedente i numeri N e M). Per gli scopi che ci prefiggiamo tutte le operazioni hanno costo unitario e si vedrà nel seguito che verranno fatte delle approssimazioni che comunque non modificheranno la validità dei risultati ottenuti.

Complessità computazionale

I teorici dell'informatica hanno introdotto delle notazioni che permettono di semplificare la rappresentazione della complessità computazionale di un algoritmo. Come primo passo il problema viene definito come funzione della grandezza dell'input, che di solito viene denotato con la lettera n. Quindi la complessità di un algoritmo viene definita come $T(n)$ dove n è la dimensione dell'input. Nell'esempio fatto in precedenza n era il valore di un numero intero, ma spesso n non rappresenta la dimensione di un singolo numero ma la cardinalità di un insieme: ad esempio nei problemi di ordinamento in cui si tratta di ordinare un insieme di oggetti secondo una certa relazione di ordinamento, la dimensione del problema è determinata dal numero di oggetti da ordinare (intuitivamente più sono gli oggetti da ordinare maggiore sarà il tempo impiegato dall'algoritmo per ordinarli). Cerchiamo di capire con un esempio: supponiamo di aver analizzato un algoritmo e aver trovato che la sua complessità computazionale valga $T(n) = n^2 + 7n + 20$, cioè tornando all'esempio del problema di ordinamento, se dovessimo ordinare 10 oggetti l'algoritmo compierebbe $100 + 70 + 20 = 190$ istruzioni, se ne dovessimo ordinare 20 allora avremmo $400 + 140 + 20 = 560$ istruzioni e così via. Quindi la formula $T(n)$ ci permette di capire il numero di istruzioni che dovranno essere compiute per risolvere un problema avendo un input di dimensione n: pur non essendo questo direttamente traducibile in un tempo fisico, è chiaro che comunque ci consente di avere un'indicazione di massima sulla possibilità o meno di arrivare a termine di un algoritmo e ci permette di confrontare algoritmi diversi tra loro.

Rispetto a quest'ultima possibilità proviamo a prendere in considerazione un algoritmo A_1 con complessità $T_1(n) = n^2 + 7n + 20$ e un altro A_2 con complessità $T_2(n) = 100n + 400$ che siano in grado di risolvere lo stesso problema: quale dei due risulta preferibile? Se ad esempio consideriamo un input di dimensione n = 10 è evidente che il primo algoritmo è più veloce in quanto ha bisogno di 190 operazioni contro le 1400 del secondo. Se però la dimensione del problema diventasse n = 100 allora il primo algoritmo avrebbe bisogno di 10720 mentre il secondo di 10400 e diventando a questo punto il secondo il più veloce. Inoltre con $n > 100$ il secondo diventerebbe via via sempre più veloce del primo, come si può facilmente notare dal grafico in figura 1.

Notazione O grande

La notazione O grande serve appunto per fare dei confronti tra la complessità di algoritmi, semplificando e ignorando parti non fondamentali del comportamento dell'algoritmo quando la dimensione dell'input cresce. Come si è visto nell'esempio precedente l'algoritmo A_2 si comporta meglio dell'algoritmo A_1 per valori di n al di sopra di una certa soglia: questo è importante perchè solitamente quello che interessa è il comportamento per input grandi e in quel caso A_2 è meglio di A_1. Ma cosa potremmo dire se avessimo un algoritmo A_3 con complessità $T_3(n) = 2n^2 + n + 2$? Intuitivamente A_2 rimane il migliore, ma cosa dire del confronto tra A_1 e A_3? Se provassimo a calcolare i valori della complessità computazionale per valori sempre più grandi di n ci accorgeremmo che il rapporto tende a stabilizzarsi intorno al numero $\frac{1}{2}$, cioè

$$\frac{T_1(n)}{T_3(n)} = \frac{n^2 + 7n + 20}{2n^2 + n + 2} \simeq \frac{1}{2}$$

per valori di n grandi a sufficienza, cioè l'algoritmo A_3 eseguirà un numero di istruzioni circa doppio dell'algoritmo A_1per valori di n sufficientemente grandi. Questo ci porta alla seguente definizione:

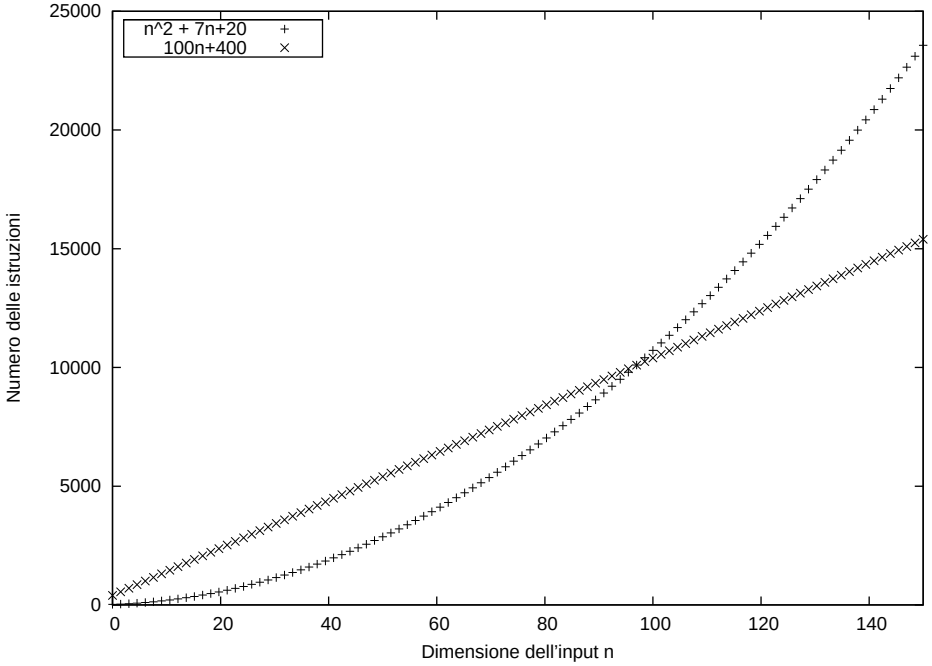

Fig. 1: Confronto tra due complessità computazionali

Definizione: Una funzione $g(n)$ è detta essere $O(f(n))$ se esistono constanti c_0 e n_0 tali che $g(n) < c_0 \cdot f(n)$ per tutti gli $n > n_0$.

Il significato di questa definizione, applicato all'esempio precedente, è che la complessità di A_3 è O(complessità di A_1) e viceversa a patto di usare degli opportuni valori di c_0 e n_0, mentre la complessità di A_3 non può essere O(complessità di A_2) in quanto non esistono valori di c_0 e n_0 tali per cui viene soddisfatta la definizione.

Questo vuol dire che mentre A_1 e A_3 nella nostra semplificazione si comportano più o meno nello stesso modo e appartengono alla classe $O(n^2)$, A_2 invece appartiene a una classe diversa denominata $O(n)$ e per valori maggiori di un certo n_0 si comporterà sempre meglio degli altri due.

Classi di complessità computazionale

Può essere utile a questo punto definire delle classi di complessità che si trovano nell'analisi degli algoritmi più utilizzati, come sono mostrate nella figura 2

Costante: gli algoritmi che appartengono a questa classe compiono sempre lo stesso numero di istruzioni indipendentemente da quanto è grande l'input e la classe viene denotata come $O(k)$.

Logaritmica: il numero delle operazioni è solitamente il logaritmo in base 2 della dimensione dell'input, poichè sono algoritmi che dividono il problema in due parti e ne risolvono solo la parte che interessa, come ad esempio la ricerca dicotomica. La classe viene denotata come $O(\lg n)$. Nella pratica non è molto diversa dalla classe costante.

Lineare: il numero delle operazioni dipende linearmente dalla grandezza dell'input, se l'input raddoppia anche il numero delle operazioni raddoppia. La classe è $O(n)$.

N log n: questa classe non ha un nome e di solito la si indica come "enne log enne". La velocità di crescita è poco più che lineare, quindi rimane comunque una classe di algoritmi "veloci". La classe è $O(n \lg n)$.

Quadratica: il numero di istruzioni cresce come il quadrato della dimensione dell'input e quindi anche per n non particolarmente grandi il numero di istruzioni può essere elevato, ma comunque trattabile. La classe è $O(n^2)$.

Cubica: in questo caso il numero di istruzioni cresce come il cubo della dimensione dell'input. In generale tutte le funzioni di crescita della forma n^k sono di tipo *polinomiale*. La classe in questo caso è $O(n^3)$.

Esponenziale: in questo caso la forma è del tipo k^n, dove n figura all'esponente e quindi la crescita del numero di istruzione è rapidissima e porta a un numero di istruzioni molto alto anche per valori piccoli di n. Nella pratica algoritmi di questo genere sono definiti *intrattabili* perchè il tempo di esecuzione risulterebbe troppo alto (anche per n piccoli si potrebbe arrivare a milioni di anni). La classe è $O(k^n)$.

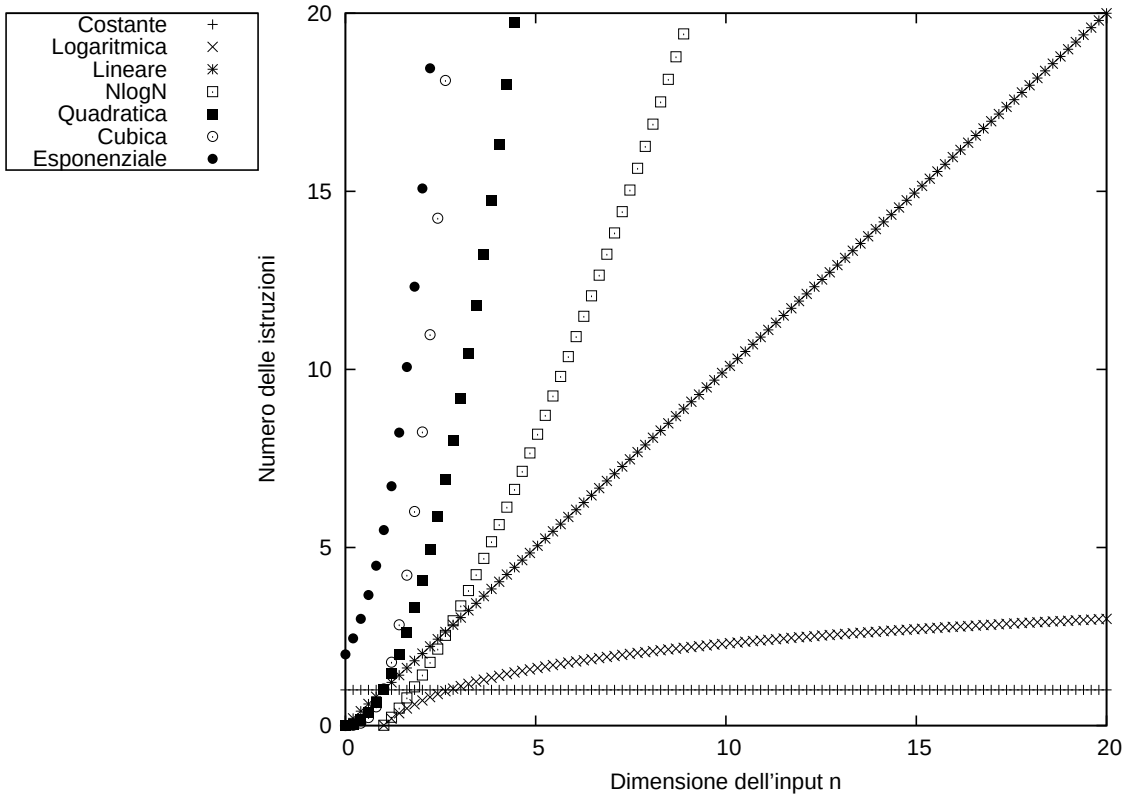

Fig. 2: Classi di complessità computazionale

Riassumendo possiamo dire che se due algoritmi fanno parte della stessa classe di complessità computazionale il loro comportamento asintotico[1] è simile. Attenzione che questo non vuol dire che in pratica non ci sia differenza tra l'algoritmo A_1 e l'algoritmo A_3 poichè ambedue appartengono a $O(n^2)$: dall'espressione della loro complessità si vede che A_1 è circa il doppio più veloce che A_3 e nella realtà questo potrebbe fare una bella

[1] Il termine asintotico ha una ben precisa definizione matematica, ma per quanto riguarda i nostri problemi possiamo approssimativamente definirlo come il comportamento quando n diventa molto grande.

differenza. Quello che si vuole ottenere con questa classificazione semplificata è solo di poter subire dire che se, ad esempio, un primo algoritmo è di classe $O(n)$ e un secondo di classe $O(n^2)$, allora il primo sarà asintoticamente molto più veloce del secondo, indipendentemente dall'espressione precisa delle loro complessità computazionali.

Caso ottimo, caso medio, caso pessimo

Finora come parametro per misurare la velocità di un algoritmo è stato usata solo la dimensione dell'input, ma nella maggior parte dei problemi ci sono altri fattori che determinano il tempo con cui un algoritmo arriva alla soluzione. Ritornando all'esempio dell'ordinamento ci sono alcuni algoritmi che sono molto veloci se l'insieme è già quasi ordinato e molto lenti se l'insieme è ordinato al contrario, pur contenendo lo stesso numero di elementi. Risulta quindi utile fare una distinzione tra caso ottimo, caso medio e caso pessimo per poter meglio valutare la bontà di un algoritmo.

Il caso ottimo è un caso particolare in cui l'algoritmo ha le prestazioni migliori e solitamente queste prestazioni sono molto migliori di quelle del caso medio e del caso pessimo. Il caso medio è quello che si manifesta più spesso in corrispondenza della maggior parte delle configurazioni dell'input. Il caso pessimo è il contrario del caso ottimo e in questo caso le prestazioni sono il peggio che si può ottenere.

Nonostante il caso ottimo possa sembrare importante, nella pratica questo è il caso meno interessante, mentre lo sono di più il caso medio e quello pessimo. Cerchiamo di capire perchè con un paio di esempi: supponiamo di dover progettare il software di un braccio meccanico per la saldatura di telai di automobili su una catena di montaggio. Un software del genere ha dei vincoli temporali molto precisi, perchè deve calcolare i movimenti in modo da saper sempre dove spostare il braccio, in modo da evitare urti o altri tipi di incidenti. Supponiamo che il tempo massimo per calcolare la nuova posizione del braccio sia di un secondo e che sappiamo che nel caso ottimo il tempo effettivo sia di un millesimo di secondo. Questa informazione ci aiuta poco perchè dice solo che in alcuni casi il nostro software riuscirà a fare molto in fretta il suo calcolo: basta però un solo caso dove il software impiega più di un secondo per distruggere il braccio o il telaio. Se invece sappiamo che nel caso pessimo il tempo di esecuzione è di 80 centesimi di secondo questo ci rassicura sul fatto che non si verificheranno mai danni per questo motivo. Anche il tempo medio è importante perchè nel caso di software in cui non sia vitale il rispetto di limiti temporali, ad esempio un word processor, sapere che nella maggior parte dei casi non supereremo un certo tempo può fare la differenza tra un software utilizzabile e uno che non lo è a causa di una latenza troppo lunga nell'esecuzione dei comandi.