A proposito di algoritmi

Giorgio Meini

Questo articolo introduce una nuova rubrica dedicata agli algoritmi ed alle loro implementazioni software. Nei prossimi mesi queste pagine ospiteranno descrizioni e realizzazioni dei più svariati tipi di algoritmi

C'era una volta, molto tempo fa, il programmatore che per mestiere si occupava di creare ed implementare algoritmi: spesso, se non sempre, si limitava ogni volta ad inventare di nuovo la ruota e, quasi sempre, si trattava di una ruota meno rotonda del dovuto! In quei tempi ormai lontani pochi erano i programmatori "di sistema" che per lavoro lottavano quotidianamente con le primitive (le API della lingua attuale) di arcaici sistemi operativi. Oggi la situazione pare essersi invertita: tutti, o quasi, i programmatori si dannano per fondere le proprie applicazioni con il sistema operativo fino al punto di renderle indistinguibili da questo e pochissimi realizzano algoritmi che sempre più frequentemente, nella forma di componenti software, anche gli altri riutilizzano. Questa trasformazione non deve sorprendere: anche la produzione del software - come già in passato è avvenuto per il mondo della microelettronica - è passata da un procedimento artigianale ad un processo standardizzato secondo criteri ingegneristici.

Per quanto molti programmi oggi prodotti siano in larga misura "autoreferenziali" rispetto all'ambiente di esecuzione (intendo dire che la maggior parte del codice di una applicazione viene scritto per gestire l'interazione con le funzionalità del sistema operativo ospite) non si può negare che il loro cuore è ancora costituito da algoritmi: l'essenza di un programma di *image processing* sono gli algoritmi di elaborazione delle immagini che rende disponibili, il resto è solo utile e, per molti motivi, indispensabile apparenza.

Questa nuova rubrica è dedicata al "cuore" dei programmi, ma non nasce per rimpiangere i tempi in cui il programmatore forgiava in proprio algoritmi tanto creativi quanto rudimentali: non sempre (quasi mai?) ciò che serve si trova in formato *componentware* (dalle antiquate librerie statiche agli oggetti distribuiti CORBA) e, ancora oggi, una abilità irrinunciabile per un "vero" programmatore consiste nella capacità di implementare algoritmi adattandoli alle proprie esigenze. Volendo realizzare ruote particolari è comunque bene renderle sufficientemente rotonde: questo è il fine che la rubrica si propone.

Limiti e potenzialità degli algoritmi

"... un algoritmo è un insieme di regole o direttive atte a fornire una risposta specifica a una specifica entrata. Caratteristica distintiva degli algoritmi è la totale eliminazione delle ambiguità: le regole devono essere sufficientemente semplici e ben definite da poter essere eseguite da una macchina. [...]. Un programma è l'esposizione di un algoritmo in un linguaggio accuratamente definito. Quindi, il programma di un calcolatore rappresenta un algoritmo, per quanto l'algoritmo stesso sia un costrutto intellettuale che esiste indipendentemente da qualsiasi rappresentazione." **[1]**.

Credo che a distanza di 20 anni dalla pubblicazione dell'articolo da cui è stata tratta la citazione questa definizione informale di algoritmo data da Donald Knuth sia ancora ampiamente condivisibile. In particolare ha il pregio di associare il concetto di algoritmo alla macchina esecutrice (il "modello di computazione") ed al linguaggio di programmazione che ne consente una definizione formale e finita. Non è necessario ricorrere alla misteriosa "macchina di Turing universale" per fornire un modello astratto di computazione: nei testi dedicati agli algoritmi l'acronimo RAM (*Random Access Machine*) designa un calcolatore ideale (si tratta di una semplice astrazione di un calcolatore sequenziale convenzionale la cui architettura è fondamentalmente basata sullo schema proposto negli anni quaranta da John Von Neumann) in cui un programma di controllo ha accesso ad una memoria illimitata composta da una sequenza di locazioni, ciascuna delle quali può contenere un numero intero arbitrario. L'universalità del modello RAM è dovuta alla rappresentazione dei programmi eseguibili in formato numerico (esattamente come avviene nei calcolatori reali), mentre il programma di controllo deve essere in grado di interpretare un linguaggio di programmazione avente potenza computazionale universale (tecnicamente definito "Turing-equivalente"): a questo scopo è sufficiente un linguaggio che implementi l'assegnazione del valore di una espressione ad una variabile, il comando di esecuzione condizionata, la nidificazione di blocchi di istruzioni da eseguirsi sequenzialmente ed il comando di esecuzione iterata (ciclo). Disponendo di un linguaggio minimale di questo tipo si è già in grado - abbandonando ogni pretesa di "potenzialità espressiva" - di definire un qualsiasi algoritmo.

Computer Programming n. 62, Ottobre 1997, pp. 84-90

La prima limitazione riguarda l'esistenza di problemi ben posti che si dimostrano essere algoritmicamente irrisolvibili. L'esempio più famoso è senz'altro il "problema dell'arresto" proposto da Alan Turing nel 1936: assegnati un generico algoritmo A ed i dati arbitrari D, decidere in tempo finito se l'esecuzione di A con *input* D termina oppure no. Con la seguente argomentazione per assurdo proposta da Fabrizio Luccio (**[2]**) si dimostra che il problema dell'arresto non è risolvibile per mezzo di un algoritmo. Infatti, se il problema fosse algoritmicamente decidibile, esisterebbe un algoritmo ARRESTO(a,d) che fornisce come *output* SI se a(d) termina e NO se a(d) non termina (ovviamente ARRESTO non potrebbe semplicemente simulare l'esecuzione a(d) perché altrimenti non potrebbe rispondere NO in tempo finito). Per l'universalità di un qualsiasi modello di computazione è sempre possibile rappresentare dati ed algoritmi utilizzando la stessa descrizione simbolica (per esempio numerica: i codici operativi delle istruzioni in linguaggio macchina di un normale calcolatore sono numeri binari!) ed è quindi possibile computare ARRESTO(A,A). Definiamo ora un ulteriore algoritmo PARADOSSO(a) in modo che esso termini nel caso che ARRESTO(a,a) fornisca come *output* NO e che non termini mai nel caso contrario. Ora PARADOSSO(PARADOSSO) termina se e solo se ARRESTO(PARADOSSO,PARADOSSO) fornisce come risultato NO, cioè se e solo se PARADOSSO(PARADOSSO) non si arresta! Una volta trovato l'assurdo nel comportamento computazionale di PARADOSSO, si deve concludere che ARRESTO non esiste e che, di conseguenza, il relativo problema è, almeno nella sua generalità, indecidibile mediante un algoritmo.

Ma la non esistenza di un algoritmo che risolva uno specifico problema non è l'unico limite da sottoporre ad analisi (anzi se questo riguarda soprattutto i logici ed i teorici, gli altri sono a carico di chi realizza un nuovo algoritmo). Scrive Paolo Zellini: "La comparsa di problemi applicativi di grandi dimensioni e l'avvento del calcolo scientifico su larga scala diedero un ulteriore contributo all'idea che i concetti teorici di risolubilità e decidibilità soffrivano di una debolezza 'pratica': la calcolabilità doveva misurarsi con le risorse di tempo e di spazio, oltre che con i limiti di precisione" **[3]**. Dato un problema non è quindi sufficiente determinare un algoritmo risolutivo, ma è "necessario che esso sia efficiente, cioè che non comporti un numero enorme di passi o un lunghissimo tempo di esecuzione." **[3]**. La misura della "complessità" degli algoritmi in termini di tempo (di esecuzione) e di spazio (di memoria) necessari per la computazione è normalmente espressa in funzione del numero N di dati forniti come *input* all'algoritmo stesso; sono convenzionalmente considerati computazionalmente "intrattabili" algoritmi che necessitano di una quantità di risorse superiore a quella esprimibile con una funzione polinomiale di N:

$$k_nN^n + k_{n-1}N^{n-1} + ... + k_iN^i + ... + k_1N + k_0.$$

Per esempio, un algoritmo che produca tutte le possibili permutazioni di una lista di N simboli deve ovviamente eseguire almeno N! operazioni ed è quindi intrinsecamente intrattabile sotto l'aspetto computazionale, mentre un algoritmo che fornisce il massimo di una lista di N valori numerici dovrà eseguire solo N operazioni di confronto ed è quindi facilmente trattabile.

In questi semplici esempi la trattabilità computazionale dipende soprattutto dalla natura del problema, ma esistono algoritmi diversi che risolvono lo stesso problema con un diverso grado di efficienza computazionale (è il caso, per esempio, dell'ordinamento di una lista di valori): al fine di scegliere la soluzione più valida è necessario disporre di tecniche di analisi che permettano di progettare e valutare rigorosamente la complessità computazionale di un algoritmo.

La scelta di considerare trattabile un algoritmo avente complessità computazionale polinomiale rispetto al numero di dati forniti come *input* può sembrare arbitraria, ma è dettata da due considerazioni ampiamente condivisibili:

- confrontando i tempi di esecuzione di 2 algoritmi che operano su N elementi eseguendo rispettivamente N^2 (complessità polinomiale) e 2^N (complessità esponenziale) operazioni, ciascuna della durata di 1 millisecondo, per $N = 2$ entrambi impiegano 4 millisecondi, ma per $N = 32$ il primo impiega 32^2=1024 millisecondi (poco più di 1 secondo), mentre il secondo elemento impiega $2^{32} = 4294967296$ millisecondi (poco meno di 50 giorni!);

- prendendo in considerazione le diverse proprietà matematiche delle funzioni polinomiali ed esponenziali si dimostra che, disponendo in futuro di calcolatori 10 volte più veloci degli attuali, a parità di tempo di esecuzione il primo algoritmo, avente complessità polinomiale N^2, tratterà un numero di dati incrementato in proporzione della radice quadrata di 10 (più del triplo: per esempio 300 anziché 100), mentre il secondo algoritmo, di complessità esponenziale 2^N, tratterà una quantità di dati aumentata del logaritmo in base 2 di 10 (poco più di tre: 103 anziché 100!).

Nel caso di algoritmi di calcolo numerico la valutazione del tempo di esecuzione e della quantità di memoria necessaria alla computazione non esaurisce l'analisi di efficienza: i calcolatori reali infatti elaborano le quantità numeriche in forma approssimata (un numero reale viene sempre rappresentato mediante una configurazione *floating-point* intera appartenente ad un intervallo limitato) e la natura "discreta" degli algoritmi impone metodi diversi rispetto alla "continuità" dei metodi matematici convenzionali; in questi casi è necessario considerare anche la "stabilità" dell'algoritmo relativamente alla propagazione ed all'amplificazione degli errori numerici.

Algoritmi sequenziali e paralleli: i modelli di computazione

Con l'avvento del nuovo millennio in molti campi di applicazione i calcolatori sequenziali saranno sostituiti da sistemi di elaborazione paralleli. La teoria degli algoritmi paralleli adotta, come modello di computazione astratto avente caratteristiche sufficientemente generali, l'architettura PRAM (*Parallel Random Access Machine*): si tratta di un insieme di N_p processori identici che condividono l'accesso ad una memoria comune globale organizzata come una sequenza potenzialmente illimitata di locazioni.

Non sempre la sintesi di un algoritmo parallelo efficace segue immediatamente dalla struttura del corrispondente algoritmo sequenziale, ma nel caso di un algoritmo sequenziale ricorsivo del tipo *divide-et-impera* (un esempio è l'algoritmo di ordinamento denominato *mergesort* **[4]**, si veda il **listato 1** per una implementazione in Pascal) si ha una trasformazione facilmente realizzabile. Caratteristica degli algoritmi di questo tipo è la suddivisione in sottoinsiemi indipendenti dell'insieme di dati forniti come *input*, a ciascuno dei quali è applicata la stessa procedura che, ricorsivamente, crea una nuova suddivisione fino a raggiungere la dimensione minima necessaria per il ricorso ad un operatore diretto; terminata la ricorsione si effettua una fusione progressiva dei risultati parziali (nel caso di ordinamento *mergesort* su PRAM si ricorre, in questa fase conclusiva, all'algoritmo parallelo di fusione di due liste ordinate noto come "metodo di Batcher" **[5]**).

L'implementazione per architettura parallela di un algoritmo ricorsivo del tipo *divide-et-impera* può realizzare la ripartizione dei dati su cui operare tra processori che eseguono contemporaneamente la stessa procedura; per esempio, se il numero di processori è sufficientemente elevato, la visita di un albero binario è realizzabile attivando, per ogni biforcazione incontrata in fase discendente, un nuovo processore secondo il criterio *breadth-first*, mentre si prosegue la visita seguendo il metodo *depth-first*: in questo modo la visita completa avviene in un tempo proporzionale alla profondità dell'albero e non al numero di nodi.

Ingenuamente si potrebbe pensare che disponendo di una PRAM con N_p processori il tempo di esecuzione di un algoritmo sequenziale diminuisca comunque di un fattore $1/N_p$, ma non è così: normalmente esistono dipendenze di sequenzialità logica tra le varie operazioni che l'algoritmo deve eseguire e quasi mai è possibile esplicitare il massimo grado di parallelismo. Un esempio estremo è rappresentato dal calcolo della potenza di un numero con il metodo delle moltiplicazioni ripetute che necessita, per ogni passo dell'algoritmo iterativo, del risultato prodotto dal passo precedente: in questa situazione il ricorso ad un calcolatore parallelo è assolutamente inutile! Anche in situazioni che non presentano questa caratteristica di sequenzialità intrinseca assoluta le necessarie operazioni di comunicazione (*broadcasting*) e di sincronismo (*collection*) tra processori diversi in fase di esecuzione di un algoritmo parallelo hanno un "costo" in termini di uso delle risorse di computazione che deve essere tenuto in considerazione **[6]**.

Il modello PRAM è una astrazione fisicamente non realizzabile (non è infatti possibile, al crescere del numero di processori, effettuare un collegamento diretto tra ciascuno di essi e la memoria comune condivisa): i calcolatori paralleli reali sono organizzati come una "rete di comunicazione" che interconnette singoli calcolatori sequenziali dotati di memoria locale; la topologia di interconnessione (*tree*, *mesh*, *hypercube*, ...; si veda in proposito la **figura 1**) caratterizza le specifiche architetture computazionali. Per quanto esistano procedure (di complessità computazionale polinomiale!) che consentono di tradurre un generico programma per PRAM in un programma per una specifica architettura, spesso si studiano algoritmi adatti ad essere eseguiti su di una particolare topologia di computazione.

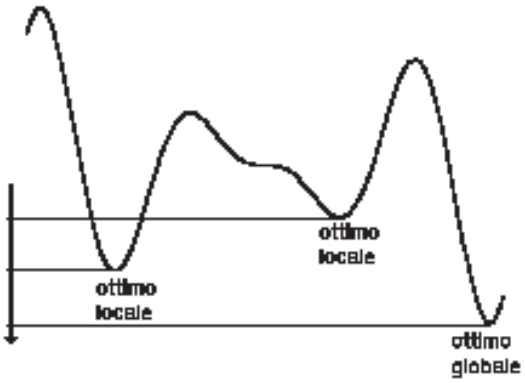

Figura 1

Definizioni degli algoritmi: i paradigmi di programmazione

Se il modello di computazione determina l'implementazione di uno specifico algoritmo, la sua definizione è fortemente influenzata dalle caratteristiche del linguaggio formale o di programmazione adottato. Tutti i linguaggi di programmazione esistenti sono espressione di uno dei seguenti "paradigmi" o di una loro combinazione **[7]**:

- imperativo
- concorrente
- orientato agli oggetti
- funzionale
- logico.

Questi paradigmi sono tutti computazionalmente equivalenti ed universali, consentono cioè di definire tutti gli algoritmi possibili ed immaginabili, ma sono ovviamente molto diversi per quanto riguarda le caratteristiche espressive; in particolare, nei linguaggi di tipo logico o funzionale, il tipico concetto procedurale di algoritmo come sequenza strutturata di operazioni da eseguire è sostituito dal concetto di predicato logico o di funzione matematica. Per un colpo d'occhio sulle conseguenze di questa diversa

"visione del mondo" si confrontino le seguenti soluzioni al classico problema delle 8 regine **[8]** - rispettivamente in Prolog (da: Sterling & Shapiro, "The art of Prolog", The MIT Press, 1986) e in Miranda (da: Tuner, "Recursion equations as a programming language", in: "Functional programming and its applications", Cambridge University Press, 1982) - con la classica soluzione ricorsiva (*backtracking*) del **listato 2**:

```
queens(N,Qs) :- range(1,N,Ns),
                permutation(Ns,Nq), safe(Qs).

safe([Q|Qs]) :- safe(Qs), not(attack(Q,Qs)).
safe([]).
attack(X,Xs) :- attack(X,1,Xs).
attack(X,N,[Y|Ys]) :- X is Y+N; X is Y-N.
attack(X,N,[Y|Ys]) :- N1 is N+1,
                              attack(X,N1,Ys).

permutation(Xs,[Z|Zs]) :- select(Z,Xs,Ys),
permutation(Ys,Zs).
                           permutation([],[]).

select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]) :- select(X,Ys,Zs).

range(M,N,[M|Ns]) :- M<N, M1 is M+1,
range(M1,N,Ns).
                                range(N,N,[N]).
```

```
queens 0 = [[]]
queens n = [ q:b | q<-[0..7]; b<-queens
                         (n-1); safe q b ], n>0
safe q b = and [ ~checks q b i | i<-[0..#b-1] ]
checks q b i = (q = b!i) \/ (abs(q - b!i) = i+1)
```

L'assoluta mancanza di istruzioni da eseguire non deve meravigliare, si tratta di linguaggi "dichiarativi" (nel caso del Prolog le soluzioni si ottengono sotto forma di lista come "chiusura" del predicato aperto *queens(8,[])*, mentre in Miranda invocando la funzione *queens 8* che restituisce un insieme): d'altra parte anche un semplice foglio di calcolo come *Excel*, pur limitando il contenuto delle celle alle sole espressioni, è computazionalmente universale (la trasformazione del tempo di scansione e calcolo delle formule nello spazio delle celle contenenti riferimenti mutuamente ricorsivi lo caratterizza come formalismo "non procedurale"). Provare per credere: come esercizio "programmare" *Excel* per risolvere il problema delle 8 regine!

Algoritmi e strutture dati

Tradizionalmente l'arte della programmazione si apprende studiando l'implementazione di "algoritmi" e di "strutture dati", l'interdipendenza tra questi due aspetti è tra l'altro sancita nel titolo di uno dei libri sacri della programmazione: *Algorithms + data structures = programs* di Niklaus Wirth. Il doppio legame è dovuto, da una parte, alla necessità di organizzare in modo strutturato i dati su cui opera un algoritmo - come nel caso degli alberi binari di ricerca (cfr. **[1]**) - e, dall'altra, all'esigenza di disporre di algoritmi che "processano" aggregati di dati significativamente strutturati, per esempio i grafi.

Un esempio rappresentativo di questa seconda situazione è il classico problema noto in letteratura come "problema del commesso viaggiatore" (TSP: *Traveling Salesman Problem*, cfr. **[9]**): dato un insieme di città caratterizzato dalle possibili vie di comunicazione e dal "costo" della relativa percorrenza (presumibilmente proporzionale alla distanza) è richiesta la determinazione del percorso ciclico che comprende tutte le città una sola volta ed avente costo di percorrenza minimo. La "geometria" (o geografia?!) del problema si presta ad essere modellata per mezzo di un grafo i cui nodi rappresentano le città e gli archi non orientati - etichettati con il costo di percorrenza - la rete di comunicazione che le interconnette: risolvere una istanza di TSP significa determinare il "ciclo hamiltoniano" del grafo per il quale la somma delle etichette numeriche degli archi che costituiscono il ciclo stesso è minima rispetto ad ogni altro ciclo possibile. Al di là del modo di implementare un grafo con le strutture dati normalmente rese disponibili in un normale linguaggio di programmazione (matrici sparse, liste di adiacenza, elementi concatenati, ...) gli algoritmi noti che risolvono TSP operano sulla struttura-grafo per determinare tutte le possibili soluzioni al fine di scegliere quella ottimale in base ad un confronto dei costi associati: vedremo nel prossimo paragrafo come un algoritmo di tipo "enumerativo" come questo non possa essere considerato computazionalmente trattabile al crescere della dimensione dei dati in ingresso (rappresentata in questo caso dal numero delle città).

L'ortogonalità della relazione che intercorre tra algoritmi e strutture dati è stata magistralmente esemplificata da Alexander Stepenov con la realizzazione della *Standard Template Library* (STL) del C++: in moltissimi casi è possibile applicare il medesimo algoritmo (generico) della libreria (una funzione *template*) a tipi diversi di *container* che presentano la stessa "interfaccia" costituita da classi di tipo *iterator* **[10]**. Questo approccio è - sotto diversi aspetti - complementare rispetto al paradigma classico della programmazione *object-oriented* dove il ricorso imperativo ad implementazioni di tipi di dato astratti (ADT: *Abstract Data Type*) lega indissolubilmente (a meno di ridefinizione per ereditarietà) le procedure algoritmiche (i "metodi") alle strutture di memorizzazione dei dati, "incapsulandole" all'interno di una classe che le comprende entrambe.

Complessità computazionale degli algoritmi

Nel primo paragrafo ho mostrato come sia necessario disporre di tecniche di analisi che permettano di valutare rigorosamente la complessità computazionale di un algoritmo. A questo scopo la letteratura sull'argomento è infarcita di

"notazioni asintotiche di complessità computazionale in ordine di grandezza", il cui scopo è quello di esprimere - in ordine di grandezza, in modo da rendere la misura indipendente dallo specifico esecutore - il "consumo" da parte di un algoritmo di risorse computazionali (in genere il tempo di esecuzione) al crescere della dimensione dei dati in ingresso. Una notazione utile a questo scopo è la seguente: *O(f(n))* indica tutte le funzioni *g(n)* tali che esistano due costanti positive *c* ed *m* per cui, per ogni $n>m$, si ha che $g(n)<cf(n)$ **[4]**; in questo caso si individua un limite superiore al comportamento asintotico - cioè al crescere di *n* - della funzione che descrive il consumo di tempo dell'algoritmo. Per esempio: un algoritmo che effettua la scansione lineare di un vettore di *n* numeri per ricercarne il massimo (assoluto) eseguendo *n* volte - in tempo pressoché costante - il confronto tra elemento attuale del vettore e massimo relativo più l'incremento dell'indice del vettore ha complessità computazionale - in tempo - avente ordine di grandezza *O(n)*, mentre l'algoritmo che esegue la moltiplicazione tra 2 matrici quadrate di ordine *N* applicando la definizione (cfr. **riquadro 1**) ha complessità $O(N^3)$. L'algoritmo di scansione lineare per la ricerca del valore massimo è "ottimo", in quanto non è ovviamente possibile risolvere il problema senza analizzare tutti gli *n* elementi del vettore e la complessità dell'algoritmo proposto ($O(n)$) corrisponde alla complessità minima (limite inferiore) del problema in esame; invece nel caso dell'algoritmo "ingenuo" per la moltiplicazione di matrici abbiamo visto (cfr. **riquadro 1**) che esiste almeno un algoritmo migliore avente complessità $O(N^{\log_2 7})$. Se l'esistenza di un algoritmo che risolve un determinato problema stabilisce esplicitamente ("a posteriori") un limite superiore concreto alla complessità del problema stesso, è invece, di norma, estremamente difficile stabilire un limite inferiore altrettanto concreto. Limiti inferiori banali ovviamente esistono sempre: se si devono necessariamente esaminare *N* dati l'algoritmo relativo avrà una complessità minima *O(N)* (si noti che nel contesto della ricerca dei limiti inferiori la notazione *O(f(n))*, che rappresenta - per definizione - un limite superiore, viene utilizzata in modo informale); ciò che resta comunque difficile è la valutazione "a priori" della complessità reale dell'algoritmo risolutivo. Nel caso che il più alto limite inferiore dedotto dalle caratteristiche intrinseche del problema da risolvere sia minore dell'ordine di complessità del migliore algoritmo risolutivo noto resta aperta la domanda se esista un algoritmo ottimo non ancora scoperto, oppure se la valutazione teorica non sia troppo ottimistica.

In ogni caso la determinazione dei limiti inferiori di risoluzione dei problemi è un compito da ricercatori (e non da programmatori!): in **[11]** è riportata una dimostrazione rigorosa (basata sugli "alberi di decisione") che stabilisce pari a $O(n\log_2 n)$ l'ordine di complessità minimo per l'ordinamento di un vettore di *n* elementi distribuiti casualmente. L'algoritmo *mergesort* (**listato 1**) implementa la tecnica *divide-et-impera* del raddoppiamento ricorsivo per ordinare un vettore di *n* numeri; ipotizzando che *n* sia una potenza di 2 (questa ipotesi è in realtà facilmente superabile) si può stabilire l'ordine di complessità asintotica dell'algoritmo notando che, per ogni livello di ricorsione, l'algoritmo stesso viene applicato ad insiemi contenenti la metà dei dati originali e che il costo della fusione dei risultati parziali è proporzionale al numero dei dati trattati **[4]**:

$$\begin{aligned}
&\text{mergesort}\ (n)= \\
&= 2\,\text{mergesort}\ (n/2)+n = \\
&= 2(2\,\text{mergesort}\ (n/2^2)+n/2)+n = \\
&= 2(2(2\,\text{mergesort}\ (n/2^3)+n/2^2)+n/2)+n = \\
&\ldots. \\
&= 2^k\,\text{mergesort}\ (n/2^k)+2^{k-1}(n/2^{k-1})+\ldots+2(n/2)+n = \\
&= 2^k\,\text{mergesort}\ (1)+kn \\
&n/2^k = 1 \text{ Æ } n = 2^k \rightarrow k = \log_2 n \\
&\text{mergesort}\ (1)=1 \rightarrow \text{mergesort}\ (n)= 2^k + kn = \\
&= 2^{\log_2 n} + \log_2 n \lozenge n = n + n\log_2 n
\end{aligned}$$

Considerando unitario il costo computazionale di *merge - sort(1)* si ha che il costo complessivo dell'algoritmo è $n + n\log_2 n$, da cui deriva la complessità asintotica $O(n\log_2 n)$: l'algoritmo è ottimo in quanto la sua complessità corrisponde al limite inferiore teorico. Il fatto che un algoritmo sia asintoticamente ottimo in ordine di grandezza rispetto al tempo di esecuzione non significa che rappresenti una scelta implementativa obbligata; innanzi tutto non si deve dimenticare che si tratta di complessità asintotica: non si può escludere che, per valori della dimensione dei dati in ingresso inferiori ad uno specifico limite critico, esista un algoritmo con prestazioni computazionali migliori (per esempio è questo il motivo per cui molte implementazioni concrete dell'algoritmo di ricerca binaria di una chiave in una serie ordinata si interrompono, una volta ridotto il campo di ricerca al di sotto di una determinata dimensione critica, per applicare una procedura di scansione lineare). Alcuni algoritmi presentano, diversamente da *mergesort*, un comportamento computazionale dipendente dai dati forniti come *input*: in questi casi l'analisi deve tenere conto della distribuzione dei dati su cui l'algoritmo opera. Generalmente si distingue il "caso pessimo" (dal punto di vista del costo computazionale dell'algoritmo) dal "caso medio" stabilito con criteri statistici: per esempio l'algoritmo di ordinamento della libreria standard del C (*quick - sort*, adottato in quanto occupa, in fase di esecuzione, la metà dello spazio di memoria necessario per eseguire *mer - gesort*) è ottimo nel caso medio, ma non lo è nel caso pessimo (che, paradossalmente, si presenta quando il vettore è già ordinato!) in cui per terminare impiega un tempo di ordine $O(n^2)$ **[11]**.

Nel primo paragrafo ho motivato la scelta - ampiamente condivisa dalla comunità dei ricercatori - di considerare trattabili, sotto l'aspetto computazionale, solo gli algoritmi aventi complessità polinomiale, la cui totalità è convenzio-

nalmente indicata come classe P (si ricordi che si tratta di una valutazione asintotica, valida al crescere del numero di dati forniti come *input*: algoritmi di complessità non polinomiale sono quotidianamente applicati con successo ad insiemi di dati di piccola dimensione). Esistono problemi "intrinsecamente esponenziali" - potenzialmente intrattabili al crescere del numero dei dati - per i quali è impossibile determinare un algoritmo risolutivo polinomiale: per esempio la determinazione di tutti i possibili sottoinsiemi di un insieme di N elementi non può avere complessità inferiore a $O(2^N)$ in quanto 2^N è il numero di risultati distinti che si devono produrre (anzi: un algoritmo che risolve effettivamente il problema con complessità $O(2^N)$ è da considerarsi ottimo!). Ma non sempre per un problema è noto un algoritmo risolutivo "ottimo": se non si evidenziano vincoli dovuti alla natura del problema stesso (maggiorando la stima del limite inferiore di complessità teorica) è lecito pensare che possa esistere un algoritmo migliore, ma in mancanza di una proposta concreta è impossibile definire esattamente la situazione. Come vedremo, in questo ambito di incertezza un ruolo particolare è rivestito dalla classe dei problemi denominati NP-completi - cui appartiene il problema TSP introdotto nel paragrafo precedente - che è costituita da problemi per i quali tutti gli algoritmi risolutivi noti sono non polinomiali, ma per i quali non esiste una dimostrazione della necessità di avere complessità computazionale esponenziale. Ma andiamo per ordine: un problema è di tipo "decisionale" se l'algoritmo che lo risolve deve semplicemente fornire una risposta affermativa o negativa e non costruire una soluzione; la maggior parte dei problemi "difficili" sotto l'aspetto computazionale sono problemi di "ottimizzazione" che non si presentano naturalmente in forma decisionale, ma che generalmente possono essere trasformati in questa forma. Per esempio il problema del commesso viaggiatore (TSP) relativo ad un specifico grafo può essere riformulato per richiedere una risposta affermativa o negativa al quesito "esiste un ciclo hamiltoniano il cui costo totale è minore di C?", dove C rappresenta un determinato valore. La classe NPè costituita da tutti i problemi per i quali la validità di una ipotetica soluzione può essere "verificata" impiegando un algoritmo avente complessità polinomiale: nel caso TSP, dato un grafo etichettato con i costi dei cammini che interconnettono i nodi ed un ciclo, è senz'altro possibile verificare se si tratta realmente di un ciclo hamiltoniano definito sul grafo e se il suo costo complessivo è minore di C in un tempo polinomiale rispetto al numero di nodi del grafo stesso. Ovviamente tutti i problemi P sono anche NP (logicamente la classe Pè inclusa nella classe NP), ma ancora oggi non è noto (per quanto appaia plausibile: per moltissimi problemi NP come TSP non è mai stato trovato un algoritmo polinomiale) se esistano problemi NPche non siano P (non è mai stato dimostrato formalmente che le classi P ed NP non coincidono). L'opinione della maggioranza dei ricercatori è che esistano problemi NP che non sono P; la validità di questa congettura è fortemente sostenuta dall'esistenza della classe dei cosiddetti problemi NP-completi: si tratta di un insieme di problemi NP per i quali non si conosce un algoritmo risolutivo polinomiale, ma che sono trasformabili l'uno nell'altro applicando esclusivamente algoritmi aventi complessità polinomiale. Il numero dei problemi classificati come NP-completi cresce ogni giorno (TSP è solo un classico esempio) e la loro sostanziale "equivalenza computazionale" ha come inverosimile conseguenza che la scoperta di un algoritmo polinomiale per uno solo di questi problemi permetterebbe "automaticamente" di risolverli tutti con complessità polinomiale!

Stabilità degli algoritmi numerici

Come già accennato nel primo paragrafo, nel caso di algoritmi di calcolo numerico la valutazione della complessità computazionale non è sufficiente per qualificarne l'efficienza: le approssimazioni "inerenti" alla rappresentazione *floating-point* dei dati numerici e la propagazione degli errori dovuta alla matematica discreta e finita dei calcolatori reali (errori "algoritmici") possono alterare i risultati forniti da un algoritmo formalmente corretto fino a renderli inutilizzabili. Inoltre la natura computazionale degli algoritmi impone metodi discreti e finiti che, non rispettando la sostanziale proprietà di continuità degli enti matematici convenzionali, introducono errori di tipo "analitico". Vediamo un esempio significativo: la derivata in un punto x_0 di una funzione reale di variabile reale $f(x)$ è formalmente definita come

$$\lim_{h\to 0} \frac{f(x_0+h)-f(x_0)}{h}$$

In questo caso la discretizzazione del concetto continuo di limite, necessaria al fine di rendere numericamente calcolabile la derivata, consiste nell'assumere come approssimazione accettabile il rapporto incrementale calcolato per un particolare valore di h

$$\frac{f(x_0+h)-f(x_0)}{h}$$

introducendo un errore di tipo analitico (quantificabile ricorrendo alla formula di Taylor per lo sviluppo in serie di funzioni di $f(x)$) che è tanto più trascurabile quanto più è piccolo h. Tenendo presente che se $f(x)=x^2$ la derivata nel punto $x_0=1$ vale esattamente *2*, si osservi la seguente tabella che elenca i risultati ottenuti calcolando, per diversi valori di h, il rapporto incrementale mediante un programma codificato in linguaggio C ricorrendo a variabili di tipo *double*:

h	df/dx_0
1E-1	2.100000
1E-2	2.010000
1E-4	2.000100
1E-8	2.000000
1E-16	1.999269
1E-32	0.000000

Effettivamente al diminuire del valore di h si ottiene un risultato progressivamente affetto da errore (analitico)

minore, ma scendendo al di sotto di un determinato valore critico l'errore (algoritmico) dovuto alla matematica approssimata della macchina diviene preponderante fino a produrre risultati inservibili.
Alcune tecniche per la realizzazione di algoritmi "numericamente stabili" sono esaminate in **[12]**, ma una delle più comuni consiste nel ricorso ad algoritmi "iterativi" che producono il risultato per approssimazioni successive. Nel caso di equazioni esprimibili nella forma $x=f(x)$ è possibile definire un metodo di risoluzione iterativo nella forma $x_{i+1}=f(x_i)$ che, in alcune condizioni, converge progressivamente al valore della soluzione.
Per esempio, volendo risolvere l'equazione

$$e^{-x} - x = 0$$

è sufficiente riscriverla come

$$x = e^{-x}$$

per definire il metodo iterativo

$$x_{i+1} = e^{-x_i}$$

implementabile con il seguente codice C

```
const int n=16;
double x=0;
int i;

for (i=0; i<n; i++)
   {
    x = exp(-x);
    printf("\n%f",x);
   }
```

che fornisce i seguenti risultati:

1.000000	0.571143
0.367879	0.564879
0.692201	0.568429
0.500474	0.566415
0.606244	0.567557
0.545396	0.566909
0.579612	0.567276
0.560115	0.567068

In questo caso il metodo "converge" al risultato corretto per un valore iniziale $x_0=0$ (in generale la convergenza di un algoritmo iterativo dipende dal valore iniziale dell'iterazione). Valutare l'efficienza di un algoritmo numerico iterativo comporta sia una analisi della stabilità rispetto alla propagazione degli errori di calcolo che una stima della velocità e delle condizioni di convergenza.

Conclusioni

In questo articolo di carattere introduttivo ho cercato di toccare - riuscendo in alcuni casi soltanto a sfiorare - i punti fondamentali di un qualsiasi discorso "a proposito di algoritmi": nei prossimi mesi la rubrica entrerà nel vivo trattando di algoritmi in carne ed ossa e molte di queste chiacchiere troveranno una collocazione più concreta. Tradizionalmente il programmatore classifica gli algoritmi in base allo "scopo" (ricerca e ordinamento di dati, calcolo numerico approssimato, riconoscimento ed interpretazione di linguaggi formali, ottimizzazione in presenza di vincoli, gestione di strutture dati, geometria computazionale, *image processing*, *pattern matching*, sistemi simbolici, simulazione deterministica o stocastica, ...) o al "tipo" (algoritmi iterativi o ricorsivi, *divide-et-impera*, *back-tracking*, programmazione dinamica, automi a stati finiti, ricerca locale, enumerazione combinatoria, ...). Questo articolo è stato scritto con l'intento di suggerire schemi complementari di classificazione degli algoritmi (modello sequenziale o parallelo di computazione, linguaggio di programmazione imperativo o dichiarativo, caratteristiche di dualità rispetto alle strutture dati, complessità computazionale, stabilità numerica, ...) che si dimostrano utili - se non indispensabili - anche nella pratica quotidiana del mestiere di programmatore.

LISTATO 1

Tratto da: N. Wirth, "Algorithms + data structures = programs", Prentice Hall, 1976

```
program eightqueens(output);
var
 i: integer;
 a: array[1..8] of boolean;
 b: array[2..16] of boolean;
 c: array[-7..7] of boolean;
 x: array[1..8] of integer;

procedure print;
 var k: integer;
begin
 for k:=1 to 8 do
  write(x[k]:4);
 writeln;
end;

procedure try(i: integer);
 var j: integer;
begin
 for j:=1 to 8 do
  if a[j] and b[i+j] and c[i-j] then
   begin
    x[i]:=j;
    a[j]:=false;
    b[i+j]:=false;
    c[i-j]:=false;
    if i<8 then
     try(i+1)
    else
     print;
    a[j]:=true;
    b[i+j]:=true;
    c[i-j]:=true;
   end
end;

begin { eightqueens }
 for i:=1 to 8 do
  a[i]:= true;
 for i:=2 to 16 do
  b[i]:= true;
 for i:=-7 to 7 do
  c[i]:= true;
 try(1);
end.
```

LISTATO 2

Tratto da: F. Luccio, E. Lodi, L. Pagli, *Algoritmica*, CNR, Pisa 1986

```
program MAIN(input,output);
const n = 10;
type vettore = array[1..n] of integer;
var A,B: vettore; i: integer; ingresso,uscita:
text;

 procedure LEGGIVETTORE(n: integer);
var k: integer;
begin
 for k:=1 to n do
  readln(ingresso,A[k])
end;

 procedure MERGE(p,r,q: integer);
var i,j,k,h: integer;
begin
 i:=p;
 k:=p;
 j:=q+1;
 while (i<=q) and (j<=r) do
  begin
   if A[i]<A[j] then
    begin
     B[k]:=A[i];
     i:=i+1
    end
   else
    begin
     B[k]:=A[j];
     j:=j+1
    end;
   k:=k+1;
  end;
 if i<=q then
  begin
   j:=r-k;
   for h:=j downto 0 do
    A[k+h]:=A[i+h]
  end;
 for j:=p to k-1 do A[j]:=b[j]
end;

 procedure MERGESORT(p,r: integer);
label 4;
var q:integer;
begin
 if p=r then
  goto 4;
 q:=(p+r) div 2;
 MERGESORT(p,q);
 MERGESORT(q+1,r);
 MERGE(p,r,q);
4:
 end;

begin { MAIN }
 assign(ingresso,'dati.in');
 assign(uscita,'dati.out');
 reset(ingresso);
 rewrite(uscita);
 LEGGIVETTORE(n);
 MERGESORT(1,n);
 for i:=1 to n do
  writeln(uscita,A[i]);
 close(uscita);
 close(ingresso);
end.
```

RIQUADRO 1
La complessità degli algoritmi per la moltiplicazione di matrici

La moltiplicazione di 2 matrici quadrate A e B di ordine N è una matrice C di ordine N il cui generico elemento è definito come

$$c_{ij} = \sum_{k=1} a_{ik} b_{kj}$$

Applicando direttamente la definizione come nella seguente funzione C

```
MatrixMultiply(float              A[N][N],float
B[N][N],float C[N][N])
{
 int i,j,k;

 for (i=1; i<=N; i++)
    for (j=1; j<=N; j++)
       {
        C[i][j] = 0;
        for (k=1; k<=N; k++)
           C[i][j] += A[i][k]*B[k][j];
    }

}
```

si eseguono N^3 moltiplicazioni.
L'algoritmo di Strassen (una classica applicazione del metodo di "raddoppiamento ricorsivo") consente di diminuire il numero di moltiplicazioni necessarie; per N=2 gli elementi della matrice C

$$c_{11} = a_{11}b_{11} + a_{12}b_{21}$$
$$c_{12} = a_{11}b_{12} + a_{12}b_{22}$$
$$c_{21} = a_{21}b_{12} + a_{22}b_{21}$$
$$c_{22} = a_{21}b_{12} + a_{22}b_{22}$$

possono essere calcolati con solo 7 moltiplicazioni (anziché 2^3=8) mediante le relazioni

$$s_1 = (a_{11} + a_{22})(b_{11} + b_{22})$$
$$s_2 = (a_{21} + a_{22})b_{11}$$
$$s_3 = a_{11}(b_{12} - b_{22})$$
$$s_4 = a_{22}(b_{21} - b_{11})$$
$$s_5 = (a_{11} + a_{22})b_{22}$$
$$s_6 = (a_{21} - a_{11})(b_{11} + b_{12})$$
$$s_7 = (a_{12} - a_{22})(b_{21} + b_{22})$$
$$c_{11} = s_1 + s_4 - s_5 + s_7$$
$$c_{12} = s_3 + s_5$$
$$c_{21} = s_2 + s_4$$

che sono applicabili anche nel caso di matrici di ordine N>2 (N pari) suddividendole in 4 sottomatrici, ciascuna di ordine N/2

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} {}_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Nel caso N sia una potenza di 2 (questa restrizione è comunque facilmente superabile) la suddivisione può essere nuovamente applicata per calcolare le moltiplicazioni matriciali espresse dalle relazioni indicate: ripetendo ricorsivamente il procedimento *divide-et-impera* fino ad ottenere matrici di ordine 2 è possibile calcolare il prodotto di 2 matrici quadrate di ordine N effettuando solo $N^{\log_2 7}$ operazioni di moltiplicazione **[12]**.

Riferimenti bibliografici

[1] D. Knuth, "Gli algoritmi", *Le Scienze* (ed. it. di *Scientific American*) n. 108, Agosto 1977

[2] F. Luccio, "Introduzione agli algoritmi paralleli", in: *Macchine e automi*, CUEN, Napoli 1995

[3] P. Zellini, "Gli algoritmi e la loro efficienza", in: *Macchine e automi*, CUEN, Napoli 1995

[4] F. Luccio, E. Lodi, L. Pagli, *Algoritmica*, CNR, Pisa 1986

[5] G. Manzini, G. Resta, "Calcolo parallelo: i problemi di fondo", in: "Matematica computazionale", *Le Scienze*, Quaderno n. 84

[6] F. Luccio, L. Pagli, "Complessità di algoritmi paralleli", in: "La matematica della complessità", *Le Scienze*, Quaderno n. 67

[7] G. Meini, "Paradigmi dei linguaggi di programmazione", *Dev.*, Novembre 1995

[8] N. Wirth, *Algorithms + data structures = programs*, Prentice Hall, 1976

[9] T. Cormen, C. Leiserson, R. Rivest, *Introduzione agli algoritmi*, vol. 3, trad. it., Jackson Libri, 1995

[10] B. Eckel, "Putting STL to work", *Unix Rewiev*, October 1996

[11] F. Luccio, *La struttura degli algoritmi*, Boringhieri, 1982

[12] R. Bevilacqua, D. Bini, M. Capovani, O. Menchi, *Introduzione alla matematica computazionale*, Zanichelli, 1987

[13] G. Meini, "Uso del generatore di numeri casuali nei programmi di simulazione", *Computer Programming*, n. 55, Febbraio 1997

[14] Goldberg, *Algorithms genetiques*, Addison Wesley, 1994

[15] Michelowitz, *Genetic algorithms + data structures = evolution programs*, 2nd ed., Springer-Verlag, 1994

[16] K. Grant, "An introduction to genetic algorithms", *C/C++ User Journal*, March 1995

[17] C. Papadimitriou, K. Steiglitz, *Combinatorial optimization: algorithms and complexity*, Prentice Hall

[18] E. Aarts, J. Korst, *Simulated annealing and Boltzmann machines*, Wiley & Sons, 1989

[19] T. Cormen, C. Leiserson, R. Rivest, *Introduzione agli algoritmi*, vol. 2, trad. it., Jackson Libri, 1994

[20] A. Jain, J. Mao, M. Mohiuddin, "Artificial neural networks: a tutorial", *IEEE Computer*, March 1996

[21] N. Serbedzija, "Simulating artificial neural networks on parallel architectures", *IEEE Comter*, March 1996

[22] J. Hopfield, D. Tank, "Neural computation of decisions in optimization problems", *Biological Cybernetics* 52 (141-152), 1985

[23] B. Angéniol, G. de La Croix Vaubois, J.Y. Le Texier, "Self-organizing maps and the travelling salesman problem", *Neural Networks* 1 (289-293), 1988

[24] M. Motta, "Introduzione alla logica ed ai sistemi fuzzy", *Computer programming* n. 56/57/58, Set./Ott./Nov. 1996

[25] T. Janzen, "C++ classes for fuzzy logic", *The C User Journal*, November 1993

[26] E. Cox, "The fuzzy systems handbook", *AP Professional*, 1994