# Conclusione

Giunti alla fine di questo testo, cerchiamo di ricavarne "la morale". Nel cercare di risolvere un problema, lo si affronta da due lati. Da una parte, si cerca di trovare costruttivamente un algoritmo che lo risolva, fornendo così una limitazione superiore alla complessità del problema. D'altro lato, si cerca di dimostrare matematicamente una limitazione inferiore alla complessità intrinseca del problema, che valga per qualsiasi algoritmo. Infine, se possibile, si cerca di individuare un algoritmo ottimo, dimostrando limitazioni inferiori via via più elevate e/o progettando algoritmi via via più veloci. In quest'ottica, le dimostrazioni di NP-completezza servono a provare rigorosamente limitazioni inferiori superpolinomiali della complessità.

In pratica, i due aspetti interagiscono e vengono entrambi affrontati dal progettista. Infatti, dati di ingresso particolarmente sfavorevoli per il comportamento di un certo algoritmo possono suggerire la strada da seguire in una dimostrazione di NP-completezza. D'altro canto, una prova di NP-completezza fallita può aiutare ad individuare particolari situazioni da affrontare nella progettazione di un algoritmo.

Una caratterizzazione matematica della soluzione spesso suggerisce un semplice algoritmo di risoluzione, di solito non molto efficiente. Applicando opportune tecniche di progettazione, delle quali le più usate sono il divide-et-impera e la programmazione dinamica, e organizzando i dati in strutture di dati appropriate, si può riuscire a migliorare notevolmente l'efficienza dell'algoritmo proposto. Queste tre fasi (caratterizzazione della soluzione, applicazione di tecniche di progettazione, strutturazione dei dati) possono essere riconsiderate più di una volta, e in un ordine qualsiasi, ottenendo versioni sempre più efficienti dell'algoritmo, la cui complessità può magari raggiungere in ordine di grandezza una limitazione inferiore dimostrata precedentemente. Trovato un algoritmo efficiente in ordine di grandezza, si prova a modificarlo "limando" se possibile le costanti coinvolte nella funzione di complessità ed effettuando prove sperimentali su dati "reali", in modo da ottenere una versione dell'algoritmo che sia efficiente non solo in teoria ma anche in pratica. Per esempio, è ben noto che il Quick Sort e l'Algoritmo del Simplesso, pur avendo complessità superiore in ordine di grandezza ad altri algoritmi che risolvono i problemi dell'Ordinamento e della Programmazione Lineare, sono efficientissimi in pratica.

Se invece il problema è "intrattabile", si possono seguire diverse strade. Innanzitutto, si può

| | *Problemi "facili" (polinomiali)* | *Problemi "difficili" (NP-ardui)* |
|---|---|---|
| 1. | DOMINO LINEARE | DOMINO LIMITATO |
| 2. | 2-SODDISFATTIBILITÀ | 3-SODDISFATTIBILITÀ |
| 3. | PROGRAMMAZIONE LINEARE | PROGRAMMAZIONE LINEARE 0/1 |
| 4. | ABBINAMENTO 2-DIMENSIONALE | ABBINAMENTO 3-DIMENSIONALE |
| 5. | CIRCUITO EULERIANO | CIRCUITO HAMILTONIANO |
| 6. | ZAINO REALE | ZAINO INTERO |
| 7. | MINIMO ALBERO DI COPERTURA | MINIMO ALBERO DI COPERTURA CON $k$ CONNESSIONI |
| 8. | CAMMINI MINIMI | CAMMINI MINIMI SENZA NODI RIPETUTI |
| 9. | 2-COLORAZIONE | 3-COLORAZIONE |

Tabella 1: Coppie di problemi apparentemente simili ma con complessità diversa.

vedere se c'è qualche caso speciale del problema che sia risolubile in tempo polinomiale. Se il problema generale è di ottimizzazione, si può tentare di individuare un algoritmo di approssimazione assoluta che abbia complessità polinomiale e fornisca una soluzione vicina a quella ottima. Se il problema non è "fortemente" NP-arduo, si può cercare un algoritmo esatto di complessità pseudopolinomiale oppure uno $\varepsilon$-approssimato di complessità polinomiale. Si possono progettare algoritmi esponenziali di tipo branch-&-bound, o ricercare algoritmi euristici, di solito basati su tecniche greedy o di ricerca locale, che siano particolarmente facili da programmare, abbiano se possibile complessità polinomiale, e forniscano sperimentalmente "buone" soluzioni su dati "reali". A tal fine, sono spesso disponibili via "internet" sia particolari dati di ingresso significativi del problema, sia il codice eseguibile e/o i risultati di vari algoritmi per lo stesso problema. È così possibile valutare la "bontà" del proprio algoritmo confrontandone, sugli stessi dati di ingresso, sia il valore della soluzione sia il tempo di calcolo con quelli di altri algoritmi già noti.

Molto importante è l'analisi dei "casi speciali" del problema. Spesso si è vittime di una inutile generalizzazione, che magari rende NP-arduo un problema altrimenti risolubile in tempo polinomiale! Se si è interessati solo ad alcuni casi speciali, l'intrattabilità del problema generalizzato è praticamente irrilevante.

Per esempio, la SODDISFATTIBILITÀ è NP-completa se in ogni clausola compaiono al più tre variabili (affermate o negate), ma è risolubile in tempo polinomiale se ne compaiono al più due! La CRICCA, l'INSIEME INDIPENDENTE, la COLORAZIONE, e il CIRCUITO HAMILTONIANO diventano polinomiali se formulati su un grafo di intervalli, i cui nodi rappresentano intervalli della retta reale e i cui archi rappresentano l'intersezione di due intervalli [cfr. Capitolo 9]. Viceversa, il MINIMO ALBERO DI COPERTURA diventa NP-completo se si richiede che ogni nodo dell'albero sia collegato ad al più $k$ nodi, con $k$ dato di input del problema. La PROGRAMMAZIONE LINEARE INTERA (anche solo 0/1) è NP-completa, ma la PROGRAMMAZIONE LINEARE, in cui le variabili $x_i$ sono reali, è risolubile in tempo polinomiale. La 3-COLORAZIONE è NP-completa, ma la 2-COLORAZIONE è risolubile in tempo polinomiale. L'ABBINAMENTO 3-DIMENSIONALE è NP-completo, ma quello 2-dimensionale (che è equivalente all'usuale ABBINAMENTO) è risolubile in tempo polinomiale.

È istruttivo osservare come molti problemi, solo in apparenza simili, abbiano complessità

| | *Problemi* | *Complessità e autori* |
|---|---|---|
| 1. | PROGRAMMAZIONE LINEARE | $O(n^{3.5}L^2)$ $L = \log\lvert\prod a_{ij}\prod b_i\rvert$<br>Karmarkar (1984) |
| 2. | FLUSSO MASSIMO | $O(mn\log(n^2/m))$<br>Goldberg-Tarjan (1986) |
| 3. | ABBINAMENTO | $O(m\sqrt{n})$<br>Micali-Vazirani (1980) |
| 4. | MINIMO ALBERO DI COPERTURA | $O(min\{m+n\log n, m\beta\})$<br>$\beta = \min\{i : \log^{(i)} n \leq m/n\}$<br>Fredman-Tarjan (1987) |
| 5. | CAMMINI MINIMI (TUTTE LE COPPIE) | $O(mn+n^2\log n)$<br>Johnson (1977) |
| 6. | MOLTIPLICAZIONE DI MATRICI | $O(n^{2.376})$<br>Coppersmith-Winograd (1987) |
| 7. | PRIMALITÀ | $O(\log^{6+\varepsilon} n)$<br>Lenstra-Pomerance (2005) |

Tabella 2: Complessità dei migliori algoritmi conosciuti per alcuni problemi.

diversa. Per esempio, il CIRCUITO HAMILTONIANO, in cui si chiede se esiste una catena chiusa che includa ogni nodo una e una sola volta, è NP-completo, mentre il CIRCUITO EULERIANO, in cui si chiede se esiste una catena chiusa che includa ogni arco una e una sola volta, è risolubile in tempo polinomiale. Analogamente, l'INSIEME INDIPENDENTE, dove si cerca un insieme di nodi a due a due non connessi da archi, è NP-arduo, mentre l'ABBINAMENTO, dove si cerca un insieme di archi a due a due non incidenti su uno stesso nodo, è risolubile in tempo polinomiale.

La Tabella 1 riassume 9 coppie di problemi apparentemente simili: quelli nella colonna di sinistra sono "facili" (risolubili in tempo polinomiale), mentre quelli nella colonna di destra sono "difficili" (NP-ardui). Alcune delle soluzioni viste in quest testo sono state scelte per la loro efficacia didattica, ma non sono gli algoritmi più efficienti in assoluto. Nella Tabella 2 riportiamo quindi alcuni di questi problemi, assieme alla complessità dei migliori algoritmi conosciuti e agli autori degli algoritmi stessi.

Infine, è bene rimarcare che lo studio dell'algoritmica non finisce qui: esistono innumerevoli altri problemi "classici", che non è stato possibile affrontare per motivi di spazio. Per alcuni di questi problemi, è possibile trovare fascicoli supplementari sul sito associato al libro.