Il più grande problema irrisolto dell'informatica

Un breve percorso intorno al concetto di complessità computazionale

Marco Liverani

5 aprile 2005

1 Premessa

Il Novecento è stato un secolo di grandi e radicali cambiamenti per le scienze e per il modo con cui gli scienziati si accostano allo studio delle rispettive discipline. Fino a buona parte dell'Ottocento lo studio delle scienze, e fra queste anche lo studio della matematica, era spesso condizionato a procedere entro i binari della ragionevolezza, dell'intuitività e della "osservabilità" dei fenomeni. Fino a tutto il 1700 la matematica era caratterizzata come lo studio dei numeri e delle figure geometriche, con alcuni settori quali l'aritmetica, l'algebra classica, l'analisi e la geometria euclidea. Nel corso del XIX secolo si svilupparono molti nuovi settori della geometria, come la geometria descrittiva, proiettiva, differenziale, non euclidea, e di altre aree della matematica, come l'analisi complessa, la teoria dei numeri, l'algebra moderna (studio di strutture come campi, anelli, gruppi, spazi vettoriali, ecc.), il calcolo delle probabilità, ecc.

In particolare cambia radicalmente il ruolo giocato dalla *teoria degli insiemi* che, a cavallo tra Ottocento e Novecento, viene ad assumere il ruolo di fondamento della matematica: la possibilità di caratterizzare insiemisticamente le funzioni, le proprietà, le relazioni, i numeri cardinali finiti e transfiniti e di definire, mediante operazioni insiemistiche, tutti gli altri tipi di numeri (interi, razionali, reali e complessi) a partire dai naturali, ha portato gradualmente ad interpretare la matematica come una disciplina fondata proprio sulla teoria degli insiemi.

Un altro punto di snodo cruciale nell'evoluzione della matematica è costituito dalla applicazione, sempre più pervasiva e rigorosa, del *metodo assiomatico*, che possiamo riassumere in questi termini. Una teoria è presentata assiomaticamente quando si assume inizialmente un certo numero di concetti come primitivi (senza definirli) e un elenco di proposizioni come assiomi (senza dimostrarli); tutti gli altri concetti della teoria vengono definiti a partire dai concetti primitivi e si dimostrano un insieme di proposizioni e teoremi a partire dagli assiomi. Nella concezione assiomatica classica (ad esempio applicata alla geometria con i famosi *Elementi* di Euclide), i concetti primitivi sono gli oggetti elementari (il punto, la retta, ecc.) e gli assiomi sono delle proprietà che appaiono intuitivamente e palesemente *vere*. In questo modo, con il procedimento rigoroso della dimostrazione, il concetto di *verità* si trasferisce dagli assiomi anche ai teoremi.

Nel corso dell'Ottocento la Matematica si è trasformata e sono potute nascere nuove teorie grazie all'assunzione di una nuova concezione dell'assiomatica: si è capito che il requisito di "veridicità" di un sistema di assiomi poteva essere sostituito dal requisito di *coerenza*. Alla base delle moderne teorie matematiche non c'è più dunque un insieme di assiomi intuitivamente veri, ma

piuttosto degli assiomi che fra loro costituiscono un sistema *coerente* e privo di contraddizioni. Gli assiomi diventano quindi delle proposizioni né vere né false, che enunciano proprietà e relazioni che si assumono senza dimostrazione. La rottura di questo vincolo ha permesso la costruzione di teorie molto più potenti e versatili di quelle classiche. Viene ad assumere la massima rilevanza il problema della coerenza, o anche della non-contradditorietà o della consistenza, di un sistema di assiomi: si accettano in matematica tutte e sole le teorie assiomatiche coerenti.

Da questi presupposti nasce anche un problema cruciale: come è possibile stabilire la consistenza di un certo sistema di assiomi?

Il matematico italiano Giuseppe Peano (1858-1932) propose una celeberrima assiomatizzazione dell'aritmetica. Ebbene, nel 1931 il logico matematico Kurt Gödel (1906-1978), a soli 23 anni, dimostrò l'*incompletezza* della teoria assiomatica dell'aritmetica, provando che esistono dei teoremi che non possono essere dedotti a partire dai soli assiomi della teoria stessa.

Lo studio della teoria assiomatica degli insiemi e i problemi di decidibilità delle teorie astratte, sfociò anche nella definizione di una nuova area di indagine scientifica, di studio e di ricerca, la *teoria della calcolabilità*. In sostanza l'assiomatizzazione delle teorie fece intravedere la possibilità di costruire strumenti automatici in grado non soltanto di eseguire autonomamente calcoli numerici complessi, ma anche di derivare rigorosamente ogni proposizione vera a partire da un sistema di assiomi coerenti. Si sviluppò quindi un ambito di ricerca molto fertile che portò in breve tempo alla nascita dei primi calcolatori elettronici. Parallelamente allo sviluppo dei computer con tecniche ingegneristiche sempre più raffinate, ed anzi anticipandone i tempi, alcuni matematici, studiosi della teoria della calcolabilità e della logica matematica, proposero dei modelli di calcolo astratti con cui affrontare lo studio dei problemi la cui soluzione potesse essere derivata sulla base di un procedimento automatico.

Tra questi, il più importante di tutti fu sicuramente l'inglese Alan Turing (1912-1954), che propose un famosissimo modello di calcolo astratto noto con il nome di *Macchina di Turing*. Si tratta di un "calcolatore ideale", dotato di un meccanismo di funzionamento piuttosto povero, programmabile in modo per certi versi primitivo, ma assai elegante, dotato di una memoria infinita (questo aspetto rende la Macchina di Turing un modello *ideale* non realizzabile in pratica).

Nel 1936 il logico matematico americano Alonzo Church avanzò l'ipotesi, nota come *Tesi di Church*, che le funzioni effettivamente calcolabili fossero tutte e sole le funzioni Turing-calcolabili; in altri termini la tesi di Church afferma che tutti i problemi la cui soluzione possa essere calcolata mediante un procedimento algoritmico, possono essere risolti anche mediante una Macchina di Turing, e viceversa. Questa ipotesi, mai dimostrata rigorosamente, ma rispetto a cui nessuno fino ad ora è riuscito a produrre un controesempio in grado di confutarne la veridicità, pone la Macchina di Turing in una posizione centrale nell'ambito della teoria degli algoritmi e della calcolabilità: stando a quanto afferma Church, infatti, la Macchina di Turing, pur così povera nella sua struttura, è potente quanto il più moderno dei computer digitali dell'ultima generazione.

La tesi di Church, letta "al contrario", afferma però anche qualcos'altro: siccome è provato che esistono dei problemi che non possono essere risolti mediante una Macchina di Turing, allora questo significa che tali problemi non possono essere risolti neanche con un algoritmo.

Come nell'ambito della matematica è stato di mostrato che esistono teorie incoerenti, così l'informatica deve fare i conti con l'esistenza di problemi che non possono essere risolti per via algoritmica. Pur rimanendo nell'ambito dei problemi che possono essere "calcolati in modo effettivo", ossia limitandoci a quei problemi la cui soluzione può essere ottenuta mediante un algoritmo, nelle pagine seguenti accenneremo alle difficoltà intrinseche proprie di determinati problemi, che pongono dei limiti, attualmente non superabili, alla calcolabilità delle soluzioni.

2 Algoritmi e complessità computazionale

Un *algoritmo* è un procedimento risolutivo costituito da una sequenza finita di passi elementari che, eseguita su una determinata *istanza* del problema che l'algoritmo è in grado di risolvere, termina in tempo finito dopo aver eseguito un numero finito di operazioni.

Consideriamo ad esempio il seguente problema: dato un insieme di numeri $A = \{a_1, a_2, \ldots, a_n\}$, si vuole individuare il minore. Un algoritmo risolutore per questo problema potrebbe essere il seguente:

Algoritmo **Minimo**(A)	
1.	Poni $min = a_1$
2.	Poni $i = 2$
3.	Se $a_i < min$ allora poni $min = a_i$
4.	Poni $i = i + 1$
5.	Se $i \leq n$ allora vai al passo 3, altrimenti prosegui
6.	Stampa *min*
7.	Fermati

La procedura che abbiamo appena presentato è costituita da istruzioni elementari (assegnazione di valori a variabili di memoria, confronti fra variabili, esecuzione di operazioni aritmetiche, e poco altro) e complessivamente è costituita da soli sette passi. Il problema viene risolto in tempo finito, ossia dopo aver eseguito un numero finito di operazioni. Inizialmente infatti (passo 1) si suppone che l'elemento minimo sia il primo elemento dell'insieme, a_1; utilizzando la variabile i come "contatore" con cui passare da un elemento dell'insieme al successivo, vengono presi in considerazione tutti gli elementi a_i dell'insieme A e, di volta in volta, il valore di ogni elemento viene confrontato con la variabile *min*, in cui è memorizzato il valore che fino a quel momento è risultato essere il più piccolo. Se il valore di a_i è minore di *min*, allora questo valore viene memorizzato in *min*. In questo modo, quando saranno stati esaminati tutti gli elementi dell'insieme (cioè quando risulterà $i > n$), in *min* sarà rimasto memorizzato il valore del più piccolo elemento di A. Dunque il procedimento termina dopo un numero finito di operazioni. Quindi possiamo concludere che la procedura di calcolo che abbiamo appena analizzato è effettivamente un algoritmo.

Per uno stesso problema possono esistere due o più algoritmi in grado di risolverlo in modo esatto. Come possiamo decidere quale fra questi algoritmi è il migliore? Intuitivamente possiamo dire che il miglior algoritmo per un determinato problema è quello che lo risolve utilizzando la minor quantità di "risorse" (ad esempio la minor quantità di memoria, nel caso di un algoritmo trasformato in un programma per un computer) nel minor "tempo" possibile. Ovviamente è bene specificare a cosa ci si riferisce parlando di quest'ultimo termine di paragone, il *tempo*. Non si intende infatti il tempo "cronologico" impiegato dall'esecutore per risolvere il problema utilizzando un certo algoritmo, perché altrimenti in questo modo misureremmo l'efficienza dell'esecutore (elettronico, come nel caso di un computer, o umano) e non quella dell'algoritmo. Nell'ipotesi quindi che ognuna delle operazioni elementari previste dall'algoritmo impegni l'esecutore per una certa unità di tempo astratta, il tempo di esecuzione potrà essere misurato come numero di operazioni elementari eseguite per applicare l'algoritmo nella risoluzione di una istanza del problema.

Un'*istanza* di un certo problema è costituita, in termini forse riduttivi, dal problema stesso insieme ai suoi dati: ad esempio un problema generale potrebbe essere quello che chiede di trovare gli zeri di un polinomio di grado n, $p(n) = a_0 x^n + a_1 x^{n-1} + \ldots + a_{n-1} x + a_n$. Un'istanza dello stesso problema è invece la richiesta di trovare gli zeri del polinomio di grado n=3 $p(n) = 2x^3 - 3x + 5$, ottenuto ponendo $a_0 = 2$, $a_1 = 0$, $a_2 = 3$, $a_3 = 5$.

Per il problema del calcolo degli zeri di un polinomio di grado n possiamo anche dire che l'insieme dei valori degli n coefficienti $a_0, a_1, \ldots, a_n$, costituisce l'*input* da fornire all'algoritmo

per risolvere il problema stesso. Così come i valori degli elementi a_1, a_2, …, a_n che costituiscono l'insieme A, sono l'input da fornire all'algoritmo per la ricerca del minimo, che abbiamo visto poc'anzi.

Date due diverse istanze di uno stesso problema, il medesimo algoritmo risolutore effettuerà un numero di operazioni elementari definito in funzione della "dimensione" dell'istanza da risolvere. Ad esempio per trovare il minimo su un insieme di $n = 10$ elementi, verranno eseguite circa 10 operazioni, mentre per trovare il minimo su un insieme di $n = 100$ elementi, ne verranno eseguite circa 100.

Possiamo quindi definire per ogni algoritmo una funzione $f(n)$, calcolata in funzione della dimensione n dell'input fornito all'algoritmo stesso, che esprime, per una istanza di lunghezza n, il numero $f(n)$ di operazioni effettuate per risolvere il problema.

Ad esempio nel caso del problema della ricerca del minimo sull'insiema A di n elementi, l'algoritmo **Minimo** riportato in precedenza, esegue un numero di operazioni pari a $f(n) = 3 + 3(n - 1)$. Infatti i passi 1, 2 e 6 vengono eseguiti solo una volta ciascuno (dunque 3 operazioni elementari complessivamente), mentre i passi 3, 4 e 5 vengono eseguiti un numero di volte pari a $n - 1$. Dunque complessivamente $3n$ operazioni; diremo quindi che l'algoritmo ha una *complessità computazionale* lineare nella dimensione dell'input, per esprimere il fatto che il numero di operazioni eseguite dall'algoritmo per risolvere una certa istanza del problema cresce linearmente in funzione della dimensione dell'input che caratterizza quella specifica istanza.

Naturalmente non esistono soltanto algoritmi che impiegano un "tempo" che cresce linearmente con la dimensione dell'input: esistono problemi più complicati che richiedono algoritmi di complessità maggiore, che eseguono quindi un numero di operazioni che cresce in funzione del quadrato o del cubo del numero di informazioni da elaborare. Ad esempio la soluzione del problema dell'ordinamento di un insieme (costruire una permutazione degli elementi dell'insieme in modo tale che questi risultino disposti in ordine non decrescente) richiede algoritmi di complessità quadratica, ossia, in altri termini, algoritmi che richiedono un numero di operazioni dell'ordine di n^2 per ordinare un insieme di n elementi.[1]

3 Un problema di ottimizzazione combinatoria

Fino ad ora abbiamo visto qualche esempio in cui per ottenere la soluzione del problema serve un algoritmo la cui complessità può essere espressa con una funzione *polinomiale*, del tipo n^k. Tuttavia esistono problemi più difficili che richiedono algoritmi ben più complessi per ottenere una soluzione. Costruiamoci un esempio facendo ricorso ad un classico problema di matematica discreta (o anche, di ottimizzazione combinatoria), ottenuto nell'ambito della teoria dei grafi.

Un *grafo* $G = (V,E)$ è una coppia di insiemi finiti, $V = \{v_1, v_2, \ldots, v_n\}$, detto insieme dei *vertici* del grafo, ed $E \subseteq V \times V$, detto insieme degli *spigoli* del grafo, costituito da alcune delle coppie di vertici di $V(G)$. Ad esempio consideriamo il grafo $G = (V,E)$ costituito dall'insieme V di n=5 vertici, $V = \{1, 2, 3, 4, 5\}$, ed $E = \{(1,2), (1,5), (2,3), (2,4), (2,5), (3,5), (3,4), (4,5)\}$, che può essere rappresentato come in Figura 1.

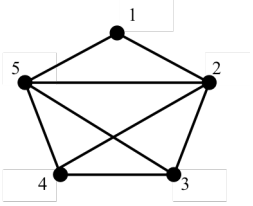

Figura 1: Un grafo G con n=5 vertici

[1] In effetti si può dimostrare che per ordinare un insieme con n elementi è possibile costruire algoritmi efficienti con una complessità dell'ordine di $n \log_2 n$.

Una *clique* su un grafo G è un sottografo G' completo massimale. Cosa sia un *sottografo* lo si può facilmente immaginare: è un grafo $G' = (V', E')$, dove $V' \subseteq V$ e $E' \subseteq E$.

Un grafo (o un sottografo) si dice *completo*, quando esiste uno spigolo per ogni coppia di vertici: ad esempio il grafo rappresentato in Figura 1 non è completo perché non esiste uno spigolo che collega la coppia di vertici 1, 4 e la coppia 1, 3.

Viceversa il sottografo G' costituito dai vertici $V' = \{1, 2, 5\}$ e dagli spigoli $E' = \{(1,2), (1,5), (2,5)\}$ è completo perché, appunto, per ogni coppia di vertici di V' esiste uno spigolo in E' che li collega.

Inoltre il sottografo G' è anche massimale. Infatti aggiungendo qualsiasi altro vertice di V a V' (e i relativi spigoli incidenti presenti in E) si perde la proprietà di completezza (ad esempio il sottografo composto dall'insieme di vertici $V'' = V' \cup \{4\}$ e dall'insieme degli spigoli $E'' = E' \cup \{(4,5), (2,4)\}$ non è un sottografo completo perché manca lo spigolo che collega i vertici 1 e 4). Dunque visto che il sottografo $G' = (V', E')$ è *completo* e *massimale*, è una *clique* di G.

Dato un grafo G vogliamo progettare un algoritmo per calcolare e stampare automaticamente tutte le *clique* di G. Il seguente algoritmo **Clique** risolve il problema.

Algoritmo **Clique**(G)
1. Sia $G = (V, E)$ il grafo ricevuto in input
2. per ogni sottoinsieme C di V ripeti le seguenti operazioni:
3. verifica se C forma un sottografo completo di G
4. verifica se C è anche massimale provando ad aggiungere gli altri vertici e gli altri spigoli
5. se entrambe le verifiche danno esito positivo allora C è una clique
6. Fermati

L'algoritmo è composto da soli 6 passi ognuno dei quali è piuttosto elementare. Tuttavia i passi 3, 4 e 5 vengono ripetuti per un numero di volte pari al numero dei sottoinsiemi di V. Ricordiamo che la cardinalità dell'*insieme delle parti* di V, $\wp(V)$, può essere calcolata facilmente: se V ha n elementi, allora l'insieme delle parti $\wp(V)$ è costituito da 2^n sottoinsiemi.

Dunque l'algoritmo **Clique**, a prescindere dal numero di operazioni elementari effettuate per eseguire ciascuno dei 6 passi (il passo 3, ad esempio, non è composto da una sola istruzione elementare, ma richiede di specificare in modo più dettagliato le operazioni da compiere), eseguirà un numero di istruzioni dell'ordine di 2^n. In questo caso la complessità dell'algoritmo viene espressa mediante una funzione non più polinomiale, ma bensì *esponenziale*, con una crescita ben più rapida: quando n aumenta di una unità, il valore della funzione $f(n) = 2^n$ raddoppia!

4 Le classi P, NP e NP-C

A prescindere dalla velocità dell'esecutore che compie le operazioni descritte dall'algoritmo (un computer, ad esempio) il tempo richiesto per eseguire una procedura di complessità polinomiale rimane sempre piuttosto contenuto, al crescere di n. Al contrario, il tempo di esecuzione richiesto da un algoritmo di complessità esponenziale cresce vertiginosamente diventando proibitivo anche per valori di n piuttosto bassi.

Ad esempio se il nostro computer esegue un milione di operazioni al secondo (per eseguire un'istruzione impiega quindi 0,000001 secondi) per eseguire un algoritmo di complessità n^2 su un'istanza del problema di dimensione $n = 10$ impiegherà appena 0,00001 secondi; se l'istanza del problema ha invece una dimensione pari a $n = 60$ lo stesso algoritmo impiegherà 0,00006 secondi, un tempo ancora decisamente trascurabile. Viceversa un algoritmo di complessità esponenziale dell'ordine di 2^n impiegherà 0,001 secondi per risolvere un'istanza del problema di dimensione $n = 10$ e circa 366 secoli (!) per risolvere un'istanza dello stesso problema con $n = 60$.

In breve, tenendo conto del comportamento nella pratica degli algoritmi risolutori, considereremo *trattabili* tutti i problemi che ammettono un algoritmo di complessità polinomiale per risolverli, mentre invece considereremo *intrattabili* i problemi, che pur essendo risolubili mediante un algoritmo, richiedono algoritmi di complessità super-polinomiale (esponenziale).

Gli studiosi di informatica teorica e di teoria della complessità e della calcolabilità, hanno suggerito una interessante classificazione per aggregare fra loro problemi anche piuttosto differenti. La prima di queste classi è denominata **P** e contiene tutti quei problemi per i quali esistono degli algoritmi risolutivi di complessità polinomiale.

Se è vero che per risolvere determinati problemi è necessario un algoritmo di complessità super-polinomiale, è anche vero che, una volta esibita una ipotetica soluzione, per alcuni di questi problemi è piuttosto facile verificare se tale soluzione sia esatta o meno. Ad esempio, è molto più facile verificare se un certo polinomio $p(x)$ si annulla in $x = 7{,}59$, piuttosto che trovare un valore di x per cui il polinomio dato assume il valore zero: verificare la correttezza di una soluzione è più facile che trovarla.

Chiameremo **NP** la classe di quei problemi che ammettono un algoritmo di complessità polinomiale in grado di verificare l'esattezza delle soluzioni. Dunque non un algoritmo *risolutore*, in grado di *trovare* la soluzione, ma un algoritmo *di verifica*, in grado di *verificare* se una certa soluzione è esatta oppure no. Se l'algoritmo di verifica ha una complessità polinomiale, allora il problema appartiene alla classe **NP**.

Naturalmente per ogni problema che può essere risolto in tempo polinomiale è possibile verificare le soluzioni in tempo polinomiale (nella peggiore delle ipotesi si applica lo stesso algoritmo progettato per *trovare* le soluzioni e si controlla se le soluzioni individuate coincidono con quelle che intendevamo verificare). Dunque $\mathbf{P} \subseteq \mathbf{NP}$.

Non è chiaro se $\mathbf{P} = \mathbf{NP}$, ossia non si sa se esistono effettivamente dei problemi che siano verificabili in tempo polinomiale, ma che non sono risolubili in tempo polinomiale (un problema del genere si troverebbe in $\mathbf{NP} - \mathbf{P}$). Fino ad oggi sono stati trovati molti problemi per i quali esiste un algoritmo di verifica di complessità polinomiale e nessun algoritmo risolutore di complessità polinomiale. Uno di questi è il problema della ricerca delle *clique* di un grafo. Tuttavia nessuno, fino ad oggi, è stato in grado di dimostrare che per quei problemi l'algoritmo risolutivo di complessità polinomiale non esiste. Dunque la questione è ancora aperta: si tratta sicuramente del problema più importante e più difficile dell'informatica, e per chi riuscirà a risolverlo, rispondendo alla domanda "**P=NP**?", un famoso centro di ricerca americano, il Clay Mathematics Institute, ha definito un premio di un milione di dollari!

A sostegno dell'ipotesi che $\mathbf{P} \neq \mathbf{NP}$, esiste una ulteriore classe di problemi, la classe dei problemi **NP-Completi** (**NPC**). L'insieme **NPC** contiene tutti quei problemi che sono **NP** e che sono tali che ogni altro problema **NP** può essere ricondotto ad essi in tempo polinomiale. Ricondurre un problema A ad un problema B significa disporre di un algoritmo T che, applicato ad ogni istanza di A, consente di ottenere una istanza di B. In questo modo con l'algoritmo risolutivo del problema B posso risolvere ogni istanza del problema A dopo averla trasformata in una istanza del problema B. Se l'algoritmo di trasformazione delle istanze di A in istanze di B è di complessità polinomiale, allora diremo che A può essere ricondotto in tempo polinomiale a B e scriveremo $A \leq_p B$.

I problemi che si trovano in **NPC** sono quindi quei problemi Q tali che $Q \in \mathbf{NP}$ e tali che, per ogni problema $Q' \in \mathbf{NP}$ risulta $Q' \leq_p Q$. La classe **NPC** costituisce quindi il "club" dei problemi **NP** più difficili. Questo insieme ha la caratteristica fondamentale che, se anche per uno solo dei suoi elementi Q (per un solo problema Q di **NPC**) venisse trovato un algoritmo risolutore di complessità polinomiale, allora si potrebbe concludere che tutti i problemi **NP** ammettono un algoritmo risolutivo di complessità polinomiale. Tale algoritmo risolutivo per risolvere un generico problema $A \in \mathbf{NP}$ lo si potrebbe ottenere attraverso la composizione dell'algoritmo polinomiale di trasformazione che consente di ricondurre ogni istanza di A in una istanza di Q, con l'algoritmo polinomiale risolutore per il problema Q.

I problemi presenti nella classe **NPC** sono numerosissimi e per nessuno di essi, dopo diversi anni di studio e di tentativi andati a vuoto, gli studiosi ed i ricercatori sono stati in grado di progettare algoritmi risolutivi di complessità polinomiale. Dunque la speranza che si possa dimostrare che **P** = **NP** diminuisce con il passare del tempo.

Riferimenti bibliografici

Testi

[1] John L. Casti, Werner DePauli, *Gödel*, Raffaello Cortina Editore, 2001 (biografia di Kurt Gödel, con una introduzione divulgativa ai suoi famosi teoremi sull'indecidibilità e l'incompletezza).

[2] Marcello Frixione, Dario Palladino, *Funzioni, macchine, algoritmi*, Carocci, 2004 (testo divulgativo, ma piuttosto rigoroso e di alto livello, sulla teoria della calcolabilità e sulle Macchine di Turing).

[3] Devlin Keith, *I problemi del millennio. I sette enigmi matematici irrisolti del nostro tempo*, Longanesi, 2004 (descrizione di taglio divulgativo dei problemi sulla cui soluzione il Clay Mathematics Institute ha posto una "taglia" di un milione di dollari).

[4] Marco Liverani, *Qual è il problema?*, Mimesis, 2005 (testo divulgativo sulla teoria degli algoritmi e della complessità computazionale).

[5] Giuliano Spirito, *Matematica senza numeri*, Newton Compton Editori, 2004 (testo divulgativo su alcuni aspetti fondamentali della teoria degli insiemi e della logica matematica).

[6] Luciana Zou, *L'informatica*, Tascabili Economici Newton, 1995 (breve testo introduttivo sugli aspetti fondamentali dell'informatica).

Risorse disponibili sulla rete Internet

[7] *Alonzo Church*, breve biografia del logico Alonzo Church e dei suoi principali lavori; a cura della Scuola di Matematica e Statistica dell'Università "St Andrews", Scozia. http://www-groups.dcs.st-and.ac.uk/~history/Mathematicians/Church.html

[8] *The Alan Turing Home Page*, riferimenti biografici e bibliografici su Alan Turing; a cura di Andrei Hodges, autore del libro *Alan Turing: the enigma.* http://www.turing.org.uk/turing/

[9] *Millennium problems*, il sito web dedicato ai problemi per i quali è in palio un premio da un milione di dollari; a cura del Clay Mathematics Institute. http://www.claymath.org/millennium

[10] *P versus NP*, la pagina del sito del Clay Mathematics Institute dedicata al problema "P=NP", tutt'ora irrisolto. http://www.claymath.org/millennium/P_vs_NP/

[11] *Grafi e alberi, introduzione*, appunti introduttivi alla teoria dei grafi. http://www.mat.uniroma3.it/users/liverani/IN1/07_grafi.pdf