1. Algoritmi e complessità computazionale

Marco Liverani

Università degli Studi Roma Tre
Dipartimento di Matematica e Fisica
Corso di Laurea in Matematica
E-mail liverani@mat.uniroma3.it

Febbraio 2014

Revisione del 23 febbraio 2014.

1 Generalità

Uno degli obiettivi principali del corso consiste nel presentare alcuni metodi per la risoluzione di problemi classici di ottimizzazione combinatoria. Per questo tipo di problemi, le cui caratteristiche possono essere meglio descritte con strumenti di carattere matematico, in generale non esistono "formule risolutive" o procedimenti basati sull'uso di strumenti matematici standard, ad esempio i tipici strumenti del calcolo o dell'analisi reale, come gli integrali o le derivate. In generale le soluzioni dei problemi che presenteremo nel corso delle lezioni, vengono costruite attraverso dei procedimenti di calcolo costituiti da una sequenza di operazioni elementari che, iterate una o più volte, consentono di produrre la soluzione di un determinato problema.

Consideriamo un esempio molto semplice: dati due numeri interi positivi x e y, vogliamo calcolarne il minimo comune multiplo, ossia il più piccolo intero che sia multiplo sia di x che di y. Una possibile strategia risolutiva di un simile problema è quella di calcolare iterativamente i multipli dell'uno e dell'altro numero fino ad ottenere un multiplo comune ad entrambi. Tale strategia può essere puntualizzata meglio delineando in modo chiaro i passi elementari da eseguire per realizzare la procedura appena descritta:

1: siano $m_x := x$ e $m_y := y$
2: se $m_x < m_y$ allora $m_x := m_x + x$ altrimenti se $m_y < m_x$ allora $m_y := m_y + y$
3: se $m_x \neq m_y$ allora vai al passo 2
4: il minimo comune multiplo tra x e y è m_x

I passi 2 e 3 della procedura vengono ripetuti più volte costruendo, a seconda dei casi, un multiplo di x, indicato con m_x, o un multiplo di y, indicato con m_y; il processo si interrompe solo quando, finalmente, sono stati raggiunti due multipli di x e di y uguali fra loro: tale multiplo comune è il risultato del problema.

Non c'è dubbio che i passi del procedimento risolutivo, considerati singolarmente, siano talmente semplici da non richiedere null'altro che capacità estremamente elementari per poter essere eseguiti: anche senza conoscere il concetto di "minimo comune multiplo", ma sapendo effettuare semplicemente delle somme e dei confronti fra numeri interi, si può risolvere il problema, attuando con pazienza e precisione i passi della procedura appena descritta.

D'altra parte, è pur vero che partendo da una procedura di calcolo descritta con passi così elementari non è facile comprenderne immediatamente la strategia risolutiva sottostante. Inoltre è necessario un certo sforzo per convincersi della correttezza della procedura: è proprio vero che il risultato calcolato attuando i passi del procedimento corrisponde al minimo comune multiplo fra x e y? Grazie al fatto che i passi sono descritti in modo elementare e tale da non lasciare spazio ad alcuna ambiguità, è possibile dimostrare la correttezza dell'algoritmo, proprio come faremmo nel provare una proprietà o un'altra affermazione matematica proposta nella tesi di un teorema.

Che il risultato calcolato dalla procedura sia un multiplo di x e di y lo si può desumere dal modo in cui il risultato m_x è stato calcolato: inizialmente $m_x = x$; successivamente il valore m_x viene modificato, sommando di volta in volta il valore x al precedente; ne segue che in qualunque momento, m_x è un multiplo di x. Lo stesso ragionamento può essere sviluppato per provare che anche m_y è un multiplo di y. D'altra parte i passi 2 e 3 vengono ripetuti più volte fino a quando m_x e m_y non avranno raggiunto il medesimo valore: in tal caso quindi $m_x = m_y$ e dunque tale valore è un multiplo comune sia ad x che ad y. Come possiamo essere certi che sia il più piccolo possibile tra tutti i multipli comuni sia ad x che ad y? Supponiamo che z sia un multiplo comune ad x e y e che z sia minore del risultato prodotto dalla procedura. Ma questo è impossibile, infatti z può essere ottenuto partendo dal valore iniziale $m_x = x$ e sommando ad m_x più volte il valore x; lo stesso può dirsi per y, visto che abbiamo supposto che z sia

anche un multiplo di y. Per cui se tale valore esistesse, sarebbe stato individuato dalla procedura che avrebbe terminato il calcolo producendo come risultato proprio il primo (e dunque il più piccolo) tra i multipli di x uguale ad uno dei multipli di y. Dunque, siccome il risultato prodotto dall'algoritmo è il minimo comune multiplo tra i due interi positivi x e y, e siccome sappiamo che tale valore esiste ed è finito per ogni coppia $x, y \in \mathbb{N}$ (sicuramente $xy \in \mathbb{N}$ è un multiplo comune ad x e y), allora abbiamo anche la certezza che la procedura di calcolo termini dopo aver eseguito un numero finito di iterazioni dei passi 2 e 3.

Naturalmente il caso preso ad esempio è talmente semplice che forse una simile dimostrazione può apparire pedante e pertanto ingiustificata. Tuttavia, dal momento che le soluzioni dei problemi che affronteremo saranno sempre ottenute mediante una procedura di calcolo descritta con una sequenza di passi elementari, è necessario provare in modo rigoroso la correttezza del procedimento di calcolo proposto, evidenziando la capacità della procedura di risolvere qualsiasi istanza del problema e non soltanto alcune istanze particolari.

Nel seguito saremo interessati a costruire esclusivamente delle buone procedure di calcolo, ossia procedimenti risolutivi costituiti da passi elementari, tali da riuscire a calcolare correttamente una soluzione per ogni istanza di un dato problema. Questa affermazione non è banale: può essere facile, infatti, cadere in errore progettando una procedura di calcolo "parziale", in grado di risolvere solo *alcune istanze* di un determinato problema. Consideriamo ad esempio la seguente procedura, che si suppone possa essere in grado di produrre la radice quadrata di un numero intero positivo x:

1: acquisisci il numero intero positivo x di cui si intende calcolare la radice quadrata
2: $i := 1$
3: se $i^2 = x$ allora scrivi il risultato i e fermati, altrimenti prosegui con il passo seguente
4: $i := i + 1$
5: torna al passo 3

La procedura, palesemente errata, adotta un approccio costruttivo che, ad un osservatore distratto, può anche apparire corretto: si calcola il quadrato dei numeri naturali $i = 1, 2, 3, \ldots$ fino ad incontrarne uno il cui quadrato sia proprio uguale ad x; quel valore è la radice quadrata di x. Se $x = 1$, $x = 4$ o $x = 9$ la procedura calcola correttamente il valore della radice quadrata di x, ma per $x = 2, 3, 5, 6, \ldots$ o, più in generale, per valori di x per i quali $\sqrt{x} \notin \mathbb{N}$, la procedura di calcolo non riuscirà ad individuare il risultato e non terminerà mai.

Attraverso questi esempi elementari abbiamo introdotto informalmente alcuni degli aspetti principali che dovranno caratterizzare le procedure di calcolo che definiremo per produrre la soluzione dei problemi affrontati nei capitoli seguenti. In particolare siamo interessati alla progettazione di procedimenti che d'ora in avanti chiameremo più precisamente *algoritmi*.

Informalmente possiamo dire che un **algoritmo** è un procedimento di calcolo caratterizzato dalle seguenti proprietà:

1. i singoli passi dell'algoritmo sono elementari: richiedono solo capacità logico-aritmetiche di base per poter essere eseguiti;

2. l'operazione descritta da ciascun passo è determinata univocamente senza alcuna ambiguità;

3. l'algoritmo è costituito da un numero finito di passi;

4. la procedura descritta dall'algoritmo termina in ogni caso dopo aver eseguito un numero finito di operazioni, producendo la soluzione del problema.

La prima proprietà (i passi dell'algoritmo devono essere *elementari*) ci assicura che il procedimento risolutivo possa essere effettivamente eseguito da chiunque e che non richieda pertanto delle capacità di calcolo straordinarie: la descrizione delle operazioni da compiere deve essere commisurata alle capacità del "modello di calcolo" che si intende adottare per eseguire l'algoritmo. Questo aspetto, come abbiamo visto con un esempio elementare nelle pagine precedenti, ci aiuta anche nel dimostrare la correttezza dell'algoritmo stesso.

La seconda caratteristica (*determinatezza* e *non ambiguità* dei passi dell'algoritmo) garantisce l'univocità e la riproducibilità del procedimento risolutivo, mettendoci al riparo da possibili differenti interpretazioni della procedura di calcolo, che potrebbero condurci ad errori irreparabili e difficilmente individuabili nella soluzione di un problema complesso.

Anche la *finitezza* dell'algoritmo è un requisito pratico assai importante: scarteremo a priori procedure di calcolo che non possano essere scritte in un numero finito di passi. Dovremo sforzarci di identificare sempre un numero finito di istruzioni ben precise con cui definire l'algoritmo, evitando di cedere alla tentazione di fare ricorso ad espressioni quali «e così via» o «eccetera...», che introdurrebbero un certo grado di indeterminazione e di scarsa chiarezza nel procedimento risolutivo.

Infine, come abbiamo visto nell'ultimo esempio, è essenziale che il procedimento di calcolo in ogni caso, per ciascuna istanza del problema che si intende risolvere, termini dopo aver eseguito un numero finito di operazioni e termini sempre producendo una soluzione del problema. Questa caratteristica, come abbiamo visto, non è scontata anche se l'algoritmo è costituito da pochi passi: la procedura proposta per il calcolo della radice quadrata è composta infatti da appena cinque passi elementari, eppure, per $x = 8$, non terminerebbe mai ed eseguirebbe un numero infinito di operazioni, iterando infinite volte i passi 3, 4 e 5. Come abbiamo già detto in precedenza ogni volta che viene costruito un nuovo algoritmo per la risoluzione di un determinato problema, è indispensabile dimostrare che tale algoritmo consenta di calcolare con un numero finito di operazioni elementari la soluzione corretta del problema, per ciascuna istanza del problema stesso.

In qualche modo, quindi, quest'ultima proprietà caratteristica degli algoritmi, introduce la necessità di effettuare un'analisi attenta del procedimento risolutivo nella sua interezza e di dimostrarne rigorosamente la correttezza facendo ricorso agli strumenti logico-formali tipici di qualunque dimostrazione matematica. Le caratteristiche precedenti, invece, potranno essere facilmente verificate esaminando singolarmente e fuori da un contesto più generale ciascun passo dell'algoritmo, senza che vi sia bisogno di entrare nel merito dell'intero procedimento risolutivo.

2 Pseudo-codifica di un algoritmo

Per descrivere in modo chiaro e non ambiguo i passi degli algoritmi, si fa spesso ricorso ad una modalità nota come "pseudo-codifica" dell'algoritmo: è una forma convenzionale per la descrizione delle operazioni, che ricalca, in modo meno rigido, ma possibilmente altrettanto preciso, il formalismo dei linguaggi di programmazione adottati per definire i passi di un programma eseguibile su un computer. In ultima analisi, infatti, si potrebbe pensare di codificare con un linguaggio di programmazione ciascun algoritmo presentato nelle pagine di queste dispense, per ottenere uno strumento di risoluzione automatica dei problemi che saranno proposti in seguito: una simile scelta renderebbe però meno immediata la comprensione degli algoritmi, appesantiti da una notazione fin troppo rigida ed arricchiti di operazioni puramente tecniche e che poco hanno a che fare con l'essenza del procedimento risolutivo che si vuole descrivere (es.: operazioni di allocazione della memoria, di definizione di variabili secondo le specifiche del linguaggio, ecc.). Pertanto descriveremo gli algoritmi adottando una "pseudo-codifica" che, pur essendo slegata da uno specifico linguaggio di programmazione,

consentirà di codificare agevolmente gli stessi algoritmi utilizzando un qualunque linguaggio di programmazione strutturata, come i linguaggi C/C++, C#, Java e Perl, solo per citarne alcuni tra i più diffusi in ambito accademico e industriale.

Naturalmente nella codifica di un programma destinato ad essere compilato ed eseguito su un computer, è necessario farsi carico di un insieme di aspetti tipici dello specifico linguaggio adottato; in particolare dovranno essere definite delle strutture dati adatte alla rappresentazione efficiente delle informazioni su cui l'algoritmo ed il programma devono operare. Questi aspetti, tipici della programmazione, esulano dai nostri obiettivi: salvo alcune eccezioni, in cui entreremo nel merito della struttura dati più adatta per la rappresentazione di un determinato oggetto complesso, le problematiche legate alla trasformazione degli algoritmi in un programma efficiente, che possa essere eseguito da un calcolatore, saranno lasciate all'abilità e alla competenza tecnica del lettore.[1]

Nella pseudo-codifica degli algoritmi innanzi tutto è bene distribuire le operazioni elementari che compongono il procedimento risolutivo su passi distinti l'uno dall'altro: questo aumenta la leggibilità dell'algoritmo e aiuta a seguire con maggiore facilità la successione delle operazioni che dovranno essere svolte per attuare la procedura di calcolo. Ciascun passo dell'algoritmo sarà numerato in modo da poter evidenziare senza alcuna ambiguità la sequenzialità delle operazioni: l'algoritmo inizia sempre dal passo 1 e, salvo differenti precisazioni contenute nelle stesse istruzioni dell'algoritmo, dopo aver eseguito il passo n dovrà essere eseguito il passo $n+1$. Al tempo stesso la numerazione progressiva delle istruzioni dell'algoritmo ci permetterà di fare riferimento ad un determinato passo semplicemente indicandone il numero progressivo con cui è identificato nella pseudo-codifica.

Nel descrivere gli algoritmi adotteremo il tipico paradigma "procedurale/imperativo": in parole povere le istruzioni presentate nei passi dell'algoritmo saranno costituite da "ordini" impartiti all'esecutore della procedura (nel caso in cui la procedura venga trasformata in un programma, l'esecutore è il calcolatore). L'operazione fondamentale è quella con cui viene assegnato un determinato valore ad una variabile: ad esempio un valore costante, il valore di un'altra variabile o il risultato di un'espressione. Indicheremo questa operazione con la notazione ":="; ad esempio con l'istruzione «$x := 3$» si indica l'operazione di assegnazione del valore 3 alla variabile x. Al primo membro di una istruzione di assegnazione deve essere sempre presente una variabile, mentre al secondo membro può esserci una qualsiasi espressione valida nel contesto in cui viene definito l'algoritmo, purché sia composta da operazioni elementari e sia priva di qualsiasi ambiguità. È sbagliato quindi scrivere «$3 := x$», perché la variabile a cui si assegna il valore deve essere riportata al primo membro, mentre l'espressione di cui si vuole calcolare il valore da assegnare alla variabile deve essere riportato a secondo membro.

L'uso dell'istruzione «$x := x+1$» è molto frequente e deve essere correttamente interpretata, per evitare di fraintenderne il significato: come si è detto con l'operatore ":=" si assegna alla variabile riportata a primo membro il valore ottenuto calcolando l'espressione riportata a secondo membro; per cui, nell'eseguire l'istruzione «$x := x+1$», si calcola il valore dell'espressione «$x+1$» ed si assegna alla variabile x il risultato di tale operazione. Naturalmente, affinché una simile istruzione abbia senso compiuto, è necessario che in uno dei passi precedenti sia stato assegnato un valore ben preciso alla variabile x, altrimenti sarà impossibile calcolare il risultato dell'operazione «$x+1$».

In generale, in un algoritmo espresso in forma "procedurale/imperativa", non ha senso riportare un'espressione aritmetica a se stante, se il risultato di tale espressione non viene contestualmente assegnato ad una variabile: è corretto quindi scrivere «$x := (y+z)/2$», mentre non ha senso scrivere soltanto «$(y+z)/2$», dal momento che non è chiaro l'uso che dovrà essere fatto del risultato di tale operazione da parte dell'esecutore dell'algoritmo.[2]

[1] A questo proposito si invita a fare riferimento a testi di informatica o di progettazione di algoritmi e strutture dati come alcuni di quelli citati in bibliografia ed in particolare [2], [3] e [10].

[2] Nella programmazione reale può capitare di imbattersi in istruzioni simili, che sono utilizzate per sfruttare effetti colla-

Nelle istruzioni con cui si definisce un algoritmo si può fare uso di altri algoritmi definiti in precedenza. Ad esempio, se l'algoritmo per il calcolo del minimo comune multiplo riportato a pagina 3 fosse denominato "mcm", allora, una volta definiti i valori delle due variabili a e b potremmo assegnare alla variabile c il valore del loro minimo comune multiplo sfruttando, come procedimento di calcolo, quello definito dall'algoritmo "mcm"; scriveremo pertanto la seguente istruzione: «$c := \text{mcm}(a,b)$». È molto importante, quindi, assegnare a ciascun algoritmo un nome ben preciso, in modo tale da poterlo "richiamare" nelle istruzioni di un altro algoritmo (o dello stesso algoritmo) utilizzandone il nome come riferimento.

Aggiungiamo, inoltre, che nella definizione di ciascun algoritmo in generale è necessario indicare i dati con cui viene caratterizzata una specifica istanza del problema da risolvere: tali informazioni costituiscono quello che in gergo informatico si definisce come l'**input** dell'algoritmo, ossia i dati che vengono forniti "dall'esterno" all'algoritmo stesso e che non possono essere costruiti autonomamente dall'algoritmo operando sugli altri dati a sua disposizione. Un algoritmo privo di dati di input effettuerà sempre la stessa operazione, senza poter generalizzare il proprio comportamento a diverse istanze di uno stesso problema. Ad esempio nel caso del problema del minimo comune multiplo tra due numeri, ogni istanza del problema è definita fornendo il valore dei due numeri naturali di cui si intende calcolare il minimo comune multiplo. Per cui, all'algoritmo risolutivo, dovranno essere forniti in input i valori da assegnare alle due variabili a e b per poter eseguire le operazioni previste dall'algoritmo stesso.

Analogamente ciascun algoritmo è caratterizzato da un **output**, ossia da uno o più dati "restituiti" dall'algoritmo come soluzione dell'istanza del problema prodotta attraverso la procedura di calcolo descritta dall'algoritmo stesso. Ogni algoritmo deve essere caratterizzato da un output, altrimenti l'esecuzione della procedura di calcolo risulterebbe completamente inutile ed ingiustificata. Facendo riferimento ancora una volta al semplice algoritmo per il calcolo del minimo comune multiplo, l'output è costituito proprio dal valore individuato come soluzione del problema nel passo 4 dell'algoritmo stesso.

Come abbiamo già visto negli esempi elementari riportati nelle pagine precedenti, un algoritmo non è composto soltanto da un elenco sequenziale di operazioni di assegnazione: la scelta delle operazioni da eseguire può variare in base al verificarsi o meno di determinate condizioni. Nella pseudo codifica di un algoritmo è possibile formulare delle espressioni logiche che potranno essere valutate e che potranno risultare vere o false: in uno dei due casi potranno essere eseguitie determinati passi dell'algoritmo, mentre, in caso contrario, saranno eseguiti dei passi differenti. Consideriamo il seguente problema elementare: dati due numeri x e y si vuole stabilire quale dei due sia il massimo. L'Algoritmo 1 propone un procedimento risolutivo elementare, basato sul confronto dei due numeri.

Le istruzioni «*se* ... *allora* ... *altrimenti* ... » rappresentano la classica struttura algoritmica **condizionale**, una delle tre strutture algoritmiche consentite dalle regole della **programmazione strutturata**. Di fatto con questa struttura, in base all'esito della valutazione di una espressione logica booleana (nell'esempio la semplice condizione «$x > y$», che, fissato il valore delle variabili x e y può essere soltanto *vera* o *falsa*, senza ammettere una terza alternativa) si stabilisce quali passi devono essere eseguiti; nell'esempio precedente, se il valore di x dovesse risultare minore o uguale a quello di y, allora, dopo aver valutato la condizione del passo 1, verrebbe eseguito il passo 4, saltando il passo 2. In ogni caso, indipendentemente quindi dall'esito della valutazione della condizione del passo 1, dopo aver eseguito il passo 2 o il passo 4, sarà eseguito il passo 6.

terali prodotti dallo specifico linguaggio nella valutazione di un'espressione; tuttavia in queste pagine manterremo un livello di generalità del tutto indipendente da uno specifico linguaggio di programmazione, per cui tali effetti collaterali non sono contemplati nella pseudo-codifica degli algoritmi che presenteremo nelle pagine seguenti.

Algoritmo 1 MASSIMO(x, y)

Input: Due numeri x e y
Output: Il massimo tra il valore di x e quello di y
1: **se** $x > y$ **allora**
2: $m := x$
3: **altrimenti**
4: $m := y$
5: **fine-condizione**
6: restituisci m

Generalmente un algoritmo esegue un numero di operazioni molto più grande del numero di passi di cui è costituito l'algoritmo stesso: il motivo sta nel fatto che spesso il calcolo della soluzione di un problema richiede che vengano eseguite *iterativamente* più volte le stesse operazioni su dati differenti. La reiterazione di determinati passi dell'algoritmo termina al verificarsi di una particolare condizione che potrà essere valutata con una specifica istruzione presente in uno dei passi dell'algoritmo. In questo modo si realizza la struttura **iterativa** prevista dalle regole della programmazione strutturata. La ripetizione di un certo blocco di istruzioni è condizionata al verificarsi o meno di una determinata condizione. Ad esempio, supponiamo di voler produrre i primi n multipli del numero naturale k. L'algoritmo risolutivo per questo semplice problema aritmetico è riportato di seguito.

Algoritmo 2 MULTIPLI(k, n)

Input: Due numeri naturali k e n
Output: I primi n multipli di k
1: $i := 1$
2: **fintanto che** $i \leq n$ **ripeti**
3: $m := i \cdot k$
4: restituisci in output il numero m
5: $i := i + 1$
6: **fine-ciclo**

I passi 3, 4 e 5 vengono ripetuti più volte fino a quando la condizione espressa nel passo 2 non diventa falsa: quando il valore di i supera quello di n (ossia, quando sono stati prodotti in output i primi n multipli di k: $k, 2k, 3k, \ldots, nk$) la reiterazione delle istruzioni dei passi 3, 4 e 5 si interrompe e l'algoritmo termina. Il blocco di istruzioni 2–6 in gergo informatico si chiama *ciclo* e l'istruzione presente nel passo 2 è la condizione che controlla la reiterazione del ciclo.

Spesso è utile evidenziare che un determinato blocco di istruzioni viene eseguito al variare di una certa variabile in un intervallo. Ad esempio volendo esprimere con un algoritmo l'espressione

$$s = \sum_{i=1}^{n} a_i$$

potremmo scrivere la procedura riportata nell'Algoritmo 3.

In tale algoritmo, al variare del valore della variabile i viene eseguita l'istruzione del passo 3: per $i = 1$ viene eseguita l'istruzione $s := s + a_1$, per $i = 2$ viene eseguita l'istruzione $s := s + a_2$ e così via fino all'ultima iterazione del ciclo, quando, per $i = n$, viene eseguita l'istruzione $s := s + a_n$. In altri termini l'istruzione presente nel passo 3 dell'algoritmo viene eseguita n volte e, al termine dell'ese-

Algoritmo 3 SOMMATORIA(n, a_1, a_2, …, a_n)

Input: Un insieme $A = \{a_i\}_{i=1,2,\ldots,n}$ e il numero di elementi di A
Output: La somma degli elementi di A
1: $s := 0$
2: **per** $i = 1, 2, \ldots, n$ **ripeti**
3: $\quad s := s + a_i$
4: **fine-ciclo**
5: restituisci s

cuzione del ciclo, nella variabile s sarà stata accumulata la somma di tutti gli elementi dell'insieme $A = \{a_1, a_2, \ldots, a_n\}$.

In alcuni casi può essere utile modificare la struttura condizionale presentata nell'Algoritmo 2, per far sì che la condizione che regola la reiterazione del ciclo venga valutata dopo aver eseguito le istruzioni del ciclo almeno una volta. Ad esempio se, dato un intero $n > 0$, volessimo verificare se si tratta di un numero pari o di un numero dispari, potremmo utilizzare il seguente algoritmo. La strategia adottata è quella di sottrarre ripetutamente 2 da n fino a raggiungere il valore 0 (nel qual caso il numero n è pari) oppure -1 (nel qual caso il numero n è dispari).

Algoritmo 4 PARI(n)

Input: Un numero naturale $n > 0$
Output: Il valore 0 se n è pari, il valore -1 se n è dispari
1: **ripeti**
2: $\quad n := n - 2$
3: **fintanto che** $n > 0$
4: restituisci n

L'istruzione presentata al passo 2 dell'algoritmo viene ripetuta fintanto che il valore di n risulta maggiore di zero: quando tale condizione diventa falsa il ciclo viene interrotto e viene eseguito il passo 4 con cui termina l'algoritmo. È bene osservare che la condizione espressa nella struttura iterativa dei passi 1–3 viene valutata *al termine* di ogni iterazione del ciclo e non all'inizio di ogni iterazione, come avveniva nel ciclo presentato nell'Algoritmo 2: questo significa che nel secondo algoritmo le istruzioni del ciclo vengono eseguite almeno una volta, anche se la condizione che regola il ciclo dovesse risultare falsa fin dall'inizio.

3 Programmazione strutturata

Nella formulazione degli algoritmi è sempre possibile attenersi alle regole della **programmazione strutturata**, che prevedono l'utilizzo delle sole tre strutture algoritmiche che abbiamo presentato nelle pagine precedenti:

- la **struttura sequenziale**, in cui le istruzioni si susseguono una di seguito all'altra;
- la **struttura condizionale**, in cui l'esito della valutazione di una condizione booleana stabilisce se debba essere eseguito un determinato insieme di istruzioni piuttosto che un altro;
- la **struttura iterativa**, in cui l'esito della valutazione di una condizione booleana stabilisce se un determinato insieme di istruzioni debba essere eseguito nuovamente.

Naturalmente le tre strutture possono essere composte fra di loro, giustapponendole una di seguito all'altra, oppure "nidificandole" una dentro l'altra; ciò che invece non è permesso dalle regole della programmazione strutturata è di "accavallare" fra loro due o più strutture, senza che una contenga completamente l'altra. Un altro vincolo imposto dalle regole della programmazione strutturata consiste nel vietare l'uso dell'istruzione di "salto incondizionato": la rigida sequenzialità nell'esecuzione dei passi dell'algoritmo può essere rotta solo mediante delle operazioni di salto sottoposte al controllo di una condizione. Infatti, utilizzando le istruzioni riportate in grassetto nella pseudo-codifica degli algoritmi presentati nelle pagine precedenti, abbiamo modificato la sequenzialità delle istruzioni (ad esempio ripetendo un certo blocco di istruzioni e quindi *saltando* indietro ad un passo già eseguito in precedenza) solo in base all'esito della valutazione di una condizione determinata dal confronto del valore di alcune delle variabili.

In questo ambito, uno dei risultati più importanti è costituito dal celeberrimo *Teorema Fondamentale della Programmazione Strutturata*, dovuto ai due matematici italiani Giuseppe Jacopini e Corrado Böhm, che afferma che ogni problema risolubile mediante un algoritmo ammette anche un algoritmo risolutivo costruito rispettando le regole della programmazione strutturata. Pertanto attenersi a tali regole non limita affatto la progettazione degli algoritmi: al contrario ci garantisce la progettazione di algoritmi eleganti e facilmente comprensibili. Inoltre, attenendoci a tali regole, saremo agevolati nel tradurre i nostri algoritmi in programmi codificati mediante un linguaggio di programmazione strutturata, come il C/C++, il Pascal o il linguaggio Java.[3]

Riepilogando, quindi, nella definizione di un algoritmo è buona norma adottare una *pseudo-codifica* basata sui seguenti principi generali:

- ogni algoritmo è connotato da un nome, mediante il quale è possibile richiamare l'algoritmo stesso in altri contesti;
- ogni algoritmo può essere caratterizzato da un insieme di informazioni fornite in input, che definiscono l'istanza del problema su cui deve operare l'algoritmo stesso;
- ogni algoritmo produce in output la soluzione dell'istanza del problema fornita in input;
- l'operazione di base su cui si articolano gli algoritmi nella forma "imperativa" è l'operazione di assegnazione, indicata con il simbolo ":=", con cui si assegna ad una variabile il valore ottenuto calcolando una determinata espressione;
- i passi dell'algoritmo sono numerati e vengono eseguiti sequenzialmente seguendo la numerazione progressiva dei passi stessi, a meno che non sia indicato qualcosa di diverso nelle istruzioni dell'algoritmo (operazioni di "salto");
- per modificare la struttura rigidamente sequenziale nella successione dei passi di un algoritmo si deve ricorrere alle strutture algoritmiche *condizionale* e *iterativa*.

La **struttura condizionale** consente di suddividere in due rami paralleli l'algoritmo: sulla base della valutazione di una condizione l'esecutore stabilirà se eseguire un ramo oppure l'altro; quando viene completata l'esecuzione dell'uno o dell'altro ramo della struttura condizionale, i due rami si ricongiungono.

Questa struttura algoritmica viene espressa mediante le seguenti istruzioni:

[3] Linguaggi *object oriented* come Java e C++ non prescindono dalle regole della programmazione strutturata nella codifica dei metodi presenti nelle classi implementate dal programma, anzi, riconducendosi ad una sintassi spesso simile a quella del linguaggio C, spingono il programmatore ad applicare le regole della programmazione strutturata.

se *condizione* **allora**
istruzioni da eseguire nel caso in cui la condizioni sia vera
altrimenti
istruzioni da eseguire nel caso in cui la condizione sia falsa
fine-condizione
istruzioni da eseguire in ogni caso, indipendentemente dall'esito della valutazione della condizione

La struttura condizionale può essere anche proposta in una forma semplificata, nel caso in cui non debbano essere eseguite operazioni specifiche quando la condizione risulta falsa:

se *condizione* **allora**
istruzioni da eseguire nel caso in cui la condizioni sia vera
fine-condizione
istruzioni da eseguire in ogni caso, indipendentemente dall'esito della valutazione della condizione

La **struttura iterativa** consente invece di ripetere più volte un determinato blocco di istruzioni: la valutazione di una condizione booleana consentirà di stabilire se il blocco di istruzioni deve essere ripetuto ancora una volta o se il "ciclo" deve essere interrotto proseguendo l'esecuzione dell'algoritmo con la prima istruzione successiva al blocco iterativo.

Come abbiamo visto negli esempi proposti nelle pagine precedenti, nella pseudo-codifica di un algoritmo si ricorre a tre forme distinte per la realizzazione di strutture algoritmiche iterative: la scelta della particolare struttura iterativa dipende da una valutazione di convenienza o di opportunità soggettiva, dal momento che non esiste una regola generale per privilegiarne una a scapito delle altre. L'unica distinzione realmente significativa consiste nella posizione, prima o dopo del blocco di istruzioni da reiterare, della condizione che controlla la ripetizione del ciclo. Le tre forme di struttura iterativa possono essere definite come segue:

1. Prima forma, la condizione *precede* il blocco di istruzioni (se la condizione è falsa fin dall'inizio, il blocco di istruzioni non viene mai eseguito):

   fintanto che *condizione* **ripeti**
   istruzioni del blocco da ripetere ciclicamente più volte
   fine-ciclo
   istruzioni da eseguire al termine della ripetizione del ciclo

2. Seconda forma, la condizione *segue* il blocco di istruzioni (il blocco di istruzioni viene eseguito almeno una volta anche se la condizione dovesse risultare falsa fin dall'inizio):

   ripeti
   istruzioni del blocco da ripetere ciclicamente più volte
   fintanto che *condizione*
   istruzioni da eseguire al termine della ripetizione del ciclo

3. Terza forma, il ciclo viene ripetuto al variare di un indice numerico intero in un determinato intervallo:

> **per** $i = 1, 2, \ldots, n$ **ripeti**
> *istruzioni del blocco da ripetere ciclicamente più volte*
> **fine-ciclo**
> *istruzioni da eseguire al termine della ripetizione del ciclo*

4 Strutture algoritmiche iterative e ricorsive

È importante osservare che la struttura iterativa, presente nella maggior parte di algoritmi, è direttamente riconducibile a quella di una funzione ricorsiva. Una funzione ricorsiva è una funzione nella cui definizione richiama se stessa. Ad esempio, il calcolo del fattoriale di un numero naturale $n > 0$ può essere espresso in termini iterativi scrivendo:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ \ldots \ \cdot 3 \cdot 2 \cdot 1$$

ossia, utilizzando il simbolo di prodotto:

$$n! = \prod_{k=1}^{n} k$$

Naturalmente risulta anche che $n! = n \cdot (n-1)!$, ossia il fattoriale di n è dato dal prodotto di n per il fattoriale del suo predecessore; per definizione $0! = 1$. Questo fatto ci permette di scrivere la funzione fattoriale in termini ricorsivi, con una definizione induttiva:

$$n! = \begin{cases} n \cdot (n-1)! & \text{se } n > 0 \\ 1 & \text{se } n = 0 \end{cases}$$

In termini algoritmici possiamo scrivere la procedura iterativa per il calcolo del fattoriale con le istruzioni presentate nell'Algoritmo 5.

Algoritmo 5 FATTORIALE(n)

```
1: f := 1
2: fintanto che n > 0 ripeti
3:     f := f · n
4:     n := n − 1
5: fine-ciclo
6: return f
```

Lo stesso algoritmo può essere rappresentato con una funzione ricorsiva utilizzando la definizione presentata poc'anzi e ottenendo così l'Algoritmo 6. Se $n > 0$ allora il calcolo del valore di $n!$ viene eseguito ricorsivamente moltiplicando n per il fattoriale di $n-1$; se invece n non è maggiore di zero, la funzione restituisce 1.

Questo procedimento di trasformazione può essere generalizzato, estendendolo a qualsiasi algoritmo espresso in forma iterativa. Ogni procedura iterativa ha una forma di questo genere:

fintanto che C **esegui** S

Algoritmo 6 FATTORIALE(n)

```
1: f := 1
2: se n > 0 allora
3:     m := n − 1
4:     f := n · FATTORIALE(m)
5: fine-condizione
6: return f
```

in cui C è la condizione che controlla la reiterazione del ciclo ed S è un insieme di istruzioni che devono essere eseguite iterativamente nel "corpo" del ciclo; l'algoritmo può quindi essere riscritto definendo una funzione ricorsiva FINTANTOCHEESEGUI(C,S):

Algoritmo 7 FINTANTOCHEESEGUI(C,S)

```
1: se C allora
2:     esegui S
3:     FINTANTOCHEESEGUI(C, S)
4: fine-condizione
```

5 Complessità computazionale

Il procedimento di calcolo proposto a pagina 4 per il calcolo della radice quadrata di un numero intero x, come abbiamo visto, è clamorosamente errato; in taluni casi, infatti, la procedura non solo non riesce ad esibire la soluzione corretta, ma non riesce neanche a terminare l'elaborazione: per determinate istanze del problema (ad esempio per $x = 3$) la procedura è destinata a non avere mai fine. Questo comportamento, effettivamente indesiderabile, è dovuto solo al fatto che la procedura proposta conteneva un errore concettuale macroscopico? Oppure esistono determinati problemi per i quali non è possibile costruire un algoritmo risolutivo?

La risposta a questo problema di fondo, che ha aperto la strada allo sviluppo di una *teoria della calcolabilità*, venne proposta da Alan Turing, un celeberrimo matematico inglese, vissuto nella prima metà del '900 e morto suicida, molto giovane, quando aveva solo 42 anni. Per la vastità, la profondità ed il livello estremamente innovativo dei suoi eccezionali contributi, Turing è considerato come uno dei padri dell'informatica.

Uno dei risultati più significativi ottenuti da Turing riguarda la progettazione di un modello di calcolo universale, astratto, ma tale da poter essere considerato equivalente ad ogni altro modello di calcolo generale. Tale modello, noto come *Macchina di Turing*, consente essenzialmente di definire in modo più rigoroso il concetto di algoritmo. L'equivalenza tra il modello di calcolo della Macchina di Turing ed altri modelli di calcolo "algoritmici" è un risultato noto come *Tesi di Church-Turing*, ottenuto in modo indipendente quasi contemporaneamente sia da Alan Turing che dal logico-matematico statunitense Alonzo Church. In modo informale possiamo dire che la Tesi di Church-Turing afferma che se è possibile calcolare la soluzione di un problema, per ogni istanza del problema stesso, allora esiste una Macchina di Turing, o un dispositivo equivalente (come un computer che esegua un programma scritto con un linguaggio di programmazione come il C o il Pascal) in grado di risolverlo, cioè di calcolarne la soluzione. Più formalmente possiamo dire che la classe delle funzioni calcolabili coincide con quella delle funzioni calcolabili da una macchina di Turing (funzioni "Turing-calcolabili").

Dunque, è possibile affermare che se esiste una funzione che non sia Turing-calcolabile, allora non è possibile definire un algoritmo in grado di calcolarla.

Sia Church che Turing giunsero, indipendentemente l'uno dall'altro, a dimostrare l'esistenza di un problema *indecidibile*, ossia di un problema la cui soluzione non può essere calcolata con i modelli di calcolo equivalenti alla Macchina di Turing. Tali argomenti meriterebbero una trattazione approfondita che però esula dagli obiettivi di questo corso. A tal proposito, invece, si rimanda ai testi [1], [4], [6], [7] e [8] citati in bibliografia. I risultati di Church e di Turing nel campo della calcolabilità ci permettono però di puntualizzare due aspetti di cui dovremo tenere conto nell'affrontare i problemi che analizzeremo in seguito:

1. non tutti i problemi possono essere risolti per via algoritmica: esistono problemi "indecidibili" per i quali non è possibile calcolare una soluzione in un tempo finito mediante un algoritmo;

2. il modello di calcolo algoritmico di cui ci siamo dotati è equivalente ad altri modelli di calcolo altrettanto potenti:[4] per questo motivo, dai procedimenti risolutivi di tipo algoritmico potremo derivare delle considerazioni di carattere generale che caratterizzano i problemi stessi, indipendentemente dallo strumento di calcolo adottato per ottenere una soluzione.

Il primo aspetto è un monito da tenere sempre presente e che ci spingerà a costruire una dimostrazione rigorosa della correttezza degli algoritmi che proporremo in seguito. Il secondo aspetto, invece, richiede qualche approfondimento necessario per comprenderne meglio il significato e la portata.

A tale proposito, si consideri un algoritmo $\mathcal{A}$ in grado di risolvere ogni istanza I di un determinato problema. Ogni istanza di un problema può essere definita elencando le informazioni necessarie a caratterizzarla completamente: la *dimensione* dell'istanza I, che indicheremo con la notazione $|I|$, è data dal numero di tali informazioni; in termini algoritmici possiamo definire l'istanza di un problema risolubile dall'algoritmo $\mathcal{A}$ come l'insieme delle informazioni che devono essere fornite in input ad $\mathcal{A}$ e che devono essere elaborate da tale algoritmo per poter produrre la soluzione dell'istanza del problema.

Per la definizione di algoritmo che abbiamo dato in precedenza, l'algoritmo sarà in grado di calcolare la soluzione dell'istanza I del problema dopo aver eseguito un numero finito t di operazioni elementari. È possibile esprimere il numero di operazioni svolte per ottenere la soluzione applicando l'algoritmo $\mathcal{A}$ in funzione della "dimensione" dell'istanza I del problema: in altri termini è possibile definire una funzione $f_{\mathcal{A}} : \mathbb{N} \to \mathbb{N}$ associata all'algoritmo $\mathcal{A}$, in grado di calcolare il numero t di operazioni svolte dall'algoritmo in base alla dimensione dell'istanza I del problema: $t = f_{\mathcal{A}}(|I|)$.

È utile osservare che nel caso in cui l'algoritmo sia eseguito da una macchina, come ad esempio un computer, possiamo ipotizzare che ciascuna operazione elementare venga eseguita in tempo costante unitario, per cui il numero t è proporzionale al tempo necessario per eseguire l'algoritmo su un'istanza I del problema. Per questo motivo spesso si considerano come misure equivalenti dell'*efficienza di un algoritmo*, il numero di operazioni svolte per risolvere un'istanza di dimensione n ed il tempo impiegato per calcolare la soluzione di tale istanza del problema.

La funzione $f_{\mathcal{A}}(n)$ con cui viene espresso il numero di operazioni svolte dall'algoritmo $\mathcal{A}$ per calcolare la soluzione di una istanza I di dimensione $|I| = n$, è chiamata **complessità computazionale** dell'algoritmo $\mathcal{A}$. Le funzioni di complessità sono funzioni discrete, non continue, definite sui numeri naturali e con valori sempre positivi: visto che la funzione esprime il numero di operazioni effettuate per eseguire un algoritmo, sarebbe ben strano se tale numero dovesse risultare negativo! Le funzioni

[4] Oltre alla Macchina di Turing, già citata in precedenza, la Tesi di Church-Turing afferma l'equivalenza tra diversi altri modelli di calcolo, quali ad esempio il *lambda calcolo*, proposto dallo stesso Church, e le funzioni ricorsive.

di complessità, in generale, sono funzioni non decrescenti: con l'aumentare delle informazioni da elaborare, non possono diminuire le operazioni da eseguire.

Dal momento che il numero di operazioni eseguite per risolvere un problema (con un algoritmo di cui sia stata dimostrata la correttezza) è finito, è sempre possibile calcolare la funzione complessità dell'algoritmo. In generale però tale operazione non è semplice: è richiesta una certa abilità e una buona dose di "rigore" logico-matematico per dimostrare che una determinata funzione f esprime effettivamente il numero di operazioni che devono essere eseguite per risolvere ogni istanza del problema mediante l'algoritmo $\mathcal{A}$. Inoltre a fronte di due istanze distinte di uno stesso problema, con la medesima dimensione, è possibile che l'algoritmo abbia due comportamenti completamente diversi, trovando più favorevole un'istanza I rispetto all'istanza $\mathcal{J}$.

Ad esempio, consideriamo il seguente problema: data una sequenza di n numeri $x_1, x_2, \ldots, x_n$, si vuole stabilire se la sequenza è in ordine crescente oppure no. Ogni istanza del problema è definita indicando gli n elementi della sequenza; dunque la dimensione dell'istanza del problema è data dal numero n. Ad esempio un'istanza I di dimensione $n = 4$ del nostro problema è data dalla sequenza $(x_1 = 37, x_2 = 12, x_3 = 15, x_4 = 22)$, mentre un'altra istanza $\mathcal{J}$ dello stesso problema, con la stessa dimensione $n = 4$, è data dalla sequenza $x_1 = 3, x_2 = 8, x_3 = 15, x_4 = 20)$. Per risolvere il problema possiamo utilizzare l'Algoritmo 8.

Algoritmo 8 VERIFICAORDINAMENTO$(x_1, x_2, \ldots, x_n)$

Input: Una sequenza di numeri $x_1, x_2, \ldots, x_n$
Output: L'algoritmo stabilisce se gli elementi della sequenza sono disposti in ordine crescente

1: $i := 1$
2: **fintanto che** $i < n$ e $x_i < x_{i+1}$ **ripeti**
3: $\quad i := i + 1$
4: **fine-ciclo**
5: **se** $i < n$ **allora**
6: $\quad$ restituisci "falso": la sequenza non è ordinata
7: **altrimenti**
8: $\quad$ restituisci "vero": la sequenza è ordinata
9: **fine-condizione**

L'algoritmo VERIFICAORDINAMENTO confronta a due a due gli elementi della sequenza iterando il ciclo descritto ai passi 2–4; è questo il cuore del procedimento risolutivo e la fase più onerosa in termini computazionali: infatti l'algoritmo eseguirà tante più operazioni, quante più volte sarà iterato questo ciclo. Dunque il calcolo della complessità dell'algoritmo si concentra sull'analisi del numero di iterazioni di tale ciclo. La condizione di arresto prevede che il valore della variabile i, inizialmente posto uguale ad 1 (passo 1), superi la soglia n, o che si individuino due elementi x_i e x_{i+1} della sequenza, che non rispettano l'ordine crescente. Pertanto se la sequenza rispetta l'ordine crescente degli elementi, come nel caso dell'istanza $\mathcal{J}$ dell'esempio, l'algoritmo eseguirà $n-1$ confronti prima di terminare il ciclo e concludere che l'intera sequenza è correttamente ordinata; al contrario nel caso dell'istanza I dell'esempio precedente, l'algoritmo si arresterà quasi subito, non appena, nella prima iterazione del ciclo, avrà verificato che i primi due elementi della sequenza x_1 e x_2 non rispettano l'ordine crescente. Questo esempio elementare mette in evidenza come due istanze di uguale dimensione dello stesso problema, possono provocare due comportamenti completamente diversi da parte di un algoritmo risolutivo: per risolvere il problema nel caso dell'istanza $\mathcal{J}$ con l'Algoritmo 8 vengono eseguite più di n operazioni elementari, mentre nel caso dell'istanza I con lo stesso algoritmo viene eseguito un numero di operazioni minore di n.

Per questo motivo, in generale, ci si occupa di studiare il comportamento dell'algoritmo, la sua complessità, nel *caso peggiore*, ossia per quelle istanze del problema meno favorevoli per l'algoritmo in esame, per la cui soluzione è richiesta l'esecuzione del massimo numero di operazioni elementari.

Inoltre, per evitare di perdere di generalità evidenziando dei dettagli del tutto irrilevanti ai fini dell'analisi della complessità di un algoritmo, spesso ci si concentra nello studio dell'ordine di grandezza della funzione complessità, per poter determinare asintoticamente il tempo richiesto per produrre una soluzione del problema, con l'aumentare dei dati forniti in input all'algoritmo. A tal fine sono state introdotte delle notazioni con cui si può indicare in modo sintetico l'ordine di grandezza di una determinata funzione di complessità computazionale. La più diffusa di tali notazioni è la cosiddetta notazione "*O* grande" (*big O*, in inglese), con cui si definisce l'insieme delle funzioni la cui crescita asintotica non è maggiore, a meno di fattori moltiplicativi costanti, a quella di una determinata funzione $f(n)$. Più precisamente si definisce la classe di funzioni $O(f(n))$ nel modo seguente:

$$O(f(n)) = \{g(n) : \exists\, c > 0, \exists\, n_0 > 0 \text{ tali che } \forall\, n > n_0 \text{ risulta } 0 \leq g(n) \leq cf(n)\} \tag{1}$$

Tali classi sono tutte non vuote, infatti per ogni $f(n)$ risulta sempre $f(n) \in O(f(n))$ per $c = 1$ e $n_0 = 0$. È utile capire però, applicando la definizione (1), quali siano le funzioni appartenenti ad una determinata classe $O(f(n))$.

Si consideri ad esempio la funzione $f(n) = n^2$ e la classe corrispondente $O(n^2)$. Quali funzioni rientrano in tale classe e quali invece non appartengono all'insieme $O(f(n))$? Come abbiamo visto $n^2 \in O(n^2)$. Inoltre ogni funzione del tipo $g(n) = \alpha n^2 + \beta n + \gamma$ appartiene alla stessa classe; infatti, per ogni funzione $g(n)$ siffatta, basterà fissare una costante $c > \alpha$ e scegliere un valore n_0 abbastanza grande tale che, per ogni $n > n_0$, risulti $(c - \alpha)n^2 > \beta n + \gamma$. Dunque $O(n^2)$ contiene tutte le funzioni quadratiche; ma contiene anche funzioni con una crescita asintoticamente meno rapida delle funzioni quadratiche, come ad esempio tutte le funzioni del tipo $g(n) = pn + q$: infatti, fissato $n_0 > 1$, è facile scegliere una costante $c > 0$ tale che $cn^2 > pn + q$ per ogni $n > n_0$. Al contrario una funzione del tipo $g(n) = \alpha n^3 + \beta n^2 + \gamma n + \delta$ non appartiene alla classe $O(n^2)$. Infatti, per quanto grande si fissi la costante $c > 0$ esisterà sempre un valore di n_0 abbastanza grande per cui risulti $cn^2 < g(n)$, per ogni $n > n_0$. La funzione $g(n) = \alpha n^3 + \ldots$ (con $\alpha > 0$) asintoticamente cresce molto più rapidamente delle funzioni cn^2, qualunque sia il valore della costante $c > 0$.

Ad esempio si consideri la funzione $f(n) = n^2$ e la funzione $g(n) = 3n^2 + 5n + 4$. Scegliendo $c = 4$, risulta $cn^2 \geq 3n^2 + 5n + 4 \implies n^2 \geq 5n + 4 \implies n \geq 5.7$; dunque $g(n) \in O(f(n))$ visto che, ponendo $c = 4$, risulta $cf(n) \geq g(n)$ per ogni $n > n_0 = 6$. Viceversa, considerando la funzione $h(n) = \frac{1}{2}n^3$ risulta $h(n) \notin O(f(n))$. Infatti, se si prova ad individuare una costante positiva c tale che $cf(n)$ domini definitivamente $h(n)$, risulta $cn^2 \geq \frac{1}{2}n^3 \implies n^2(c - \frac{1}{2}n) \geq 0 \implies c - \frac{1}{2}n \geq 0 \implies 2c \geq n$; questo significa che fissata arbitrariamente la costante $c > 0$ risulta $h(n) > cf(n)$ per ogni $n > 2c$.

In generale, è facile verificare che le funzioni "polinomiali" del tipo $g(n) = a_0 n^k + a_1 n^{k-1} + \cdots + a_{k-1} n + a_k$ appartengono alle classi $O(n^p)$, se e solo se $p \geq k$. Inoltre, per $p, q \geq 0$, $O(n^p) \subseteq O(n^q)$ se $p \leq q$.

Un'altra notazione assai importante e diffusa è la notazione $\Omega(f(n))$ con cui si definisce la classe delle funzioni il cui limite inferiore asintotico è dato da $cf(n)$, per qualche costante $c > 0$; formalmente possiamo scrivere:

$$\Omega(f(n)) = \{g(n) : \exists\, c > 0, \exists\, n_0 > 0 \text{ tali che } \forall\, n > n_0 \text{ risulta } cf(n) \leq g(n)\} \tag{2}$$

Un'ulteriore importante notazione per identificare determinate classi di funzioni è costituita da $\Theta(f(n))$: in questo caso le funzioni appartenenti a tale classe sono tutte quelle che hanno lo stesso

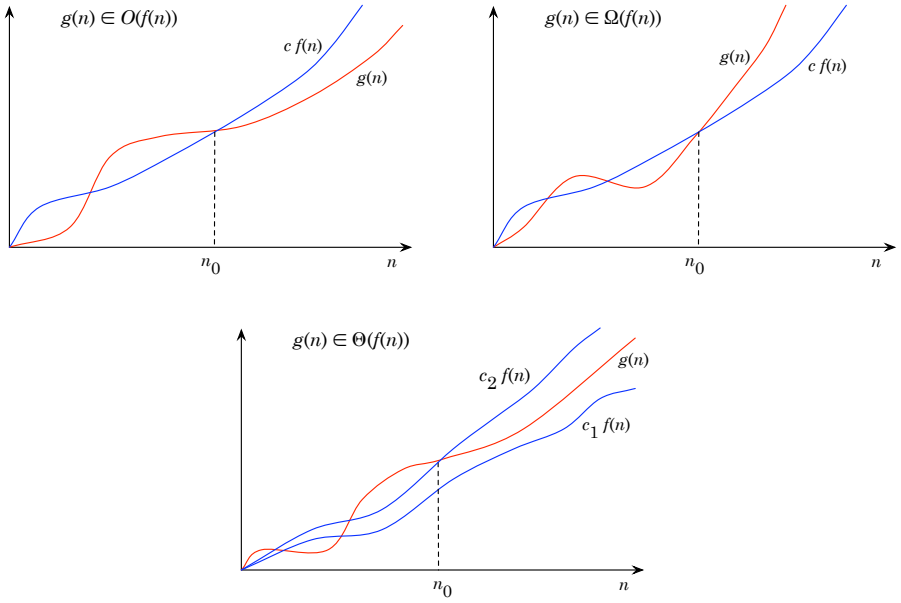

Figura 1: Esemplificazione grafica della notazione asintotica $O(f(n))$, $\Omega(f(n))$ e $\Theta(f(n))$.

andamento asintotico di $f(n)$:

$$\Theta(f(n)) = \{g(n) : \exists\, c_1, c_2 > 0, \exists\, n_0 > 0 \text{ tali che } \forall\, n > n_0 \text{ risulta } c_1 f(n) \leq g(n) \leq c_2 f(n)\} \qquad (3)$$

In Figura 1 sono rappresentati i grafici di tre funzioni $g(n)$ appartenenti alle classi $O(f(n))$, $\Omega(f(n))$ e $\Theta(f(n))$. Dalle definizioni si ricavano facilmente le proprietà raccolte nella seguente Proposizione.

Proposizione 1. *Date due funzioni $f(n)$ e $g(n)$ valgono le seguenti relazioni:*

1. $g(n) \in O(f(n))$ *se e solo se* $f(n) \in \Omega(g(n))$*;*
2. $g(n) \in \Theta(f(n))$ *se e solo se* $g(n) \in O(f(n))$ *e* $g(n) \in \Omega(f(n))$*;*
3. $g(n) \in \Theta(f(n))$ *se e solo se* $f(n) \in \Theta(g(n))$.

Dimostrazione. La dimostrazione delle tre affermazioni è piuttosto elementare.

1. Dopo aver osservato che le definizioni di $O(f(n))$ e di $\Omega(f(n))$ godono di una certa simmetria, supponiamo che $g(n) \in O(f(n))$; dunque esistono $n_0 > 0$ e $c > 0$ tali che $g(n) \leq cf(n)$ per ogni $n > n_0$; pertanto, ponendo $c' = 1/c$ si ha che $f(n) \geq c'g(n)$ per ogni $n > n_0$. Per cui $f(n) \in \Omega(g(n))$. Analogamente si può dimostrare l'implicazione inversa.

2. Supponiamo che $g(n) \in \Theta(f(n))$, per cui esistono $n_0 > 0$ e due costanti $c_1, c_2 > 0$ tali che $c_1 f(n) \leq g(n)$ e, al tempo stesso, $g(n) \leq c_2 f(n)$ per ogni $n > n_0$. Quindi, per la prima delle due disuguaglianze, risulta $g(n) \in \Omega(f(n))$, mentre per la seconda risulta $g(n) \in O(f(n))$.

   Al contrario se risulta che $g(n) \in \Omega(f(n))$, allora esistono $n_0' > 0$ e $c' > 0$ tali che $c' f(n) \leq g(n)$ per ogni $n > n_0'$. Se inoltre $g(n) \in O(f(n))$, allora esistono $n_0'' > 0$ e $c'' > 0$ tali che $g(n) \leq c'' f(n)$ per ogni $n > n_0''$. Dunque, scegliendo $n_0 = \max(n_0', n_0'')$ si ha che $c' f(n) \leq g(n) \leq c'' f(n)$ per ogni $n > n_0$. Quindi $g(n) \in \Theta(f(n))$.

3. Se $g(n) \in \Theta(f(n))$ allora per la proprietà 2 appena dimostrata risulta che $g(n) \in O(f(n))$ e $g(n) \in \Omega(f(n))$; per la proprietà 1, risulta che $g(n) \in O(f(n)) \implies f(n) \in \Omega(g(n))$ e $g(n) \in \Omega(f(n)) \implies f(n) \in O(g(n))$. Quindi per la proprietà 2 risulta $f(n) \in \Theta(g(n))$.

□

Naturalmente non tutte le funzioni che esprimono la complessità computazionale di un algoritmo sono polinomiali. Ad esempio è possibile costruire algoritmi di complessità costante $f(n) = c$ o logaritmica, del tipo $f(n) = \log_2 n$, e quindi più in generale algoritmi con complessità *sub-lineare*. Si tratta di algoritmi che, fatta eccezione per la fase di acquisizione in input dei dati che caratterizzano l'istanza del problema, eseguono meno di n operazioni per risolvere il problema stesso. È chiaro infatti che ciascun algoritmo risolutivo dovrà per lo meno acquisire in input tutte le n informazioni che caratterizzano l'istanza del problema: per cui almeno n operazioni dovranno essere effettuate; tuttavia, focalizzandosi soltanto sulla procedura con si calcola la soluzione del problema, scorporandola dalla fase di acquisizione dei dati in input, alcuni algoritmi possono essere talmente efficienti da richiedere l'esecuzione di un numero molto basso di operazioni elementari, inferiore alla dimensione stessa dell'istanza del problema.

Un esempio banale di algoritmo di complessità costante, potrebbe essere quello che, data una sequenza composta da n numeri, restituisce un elemento qualsiasi della sequenza, ad esempio il primo. Possiamo definire invece un esempio non banale di algoritmo sub-lineare di complessità logaritmica. Si consideri una sequenza $S = (x_1, \ldots, x_n)$ di numeri in ordine crescente: $x_1 < x_2 < \cdots < x_n$; fissato arbitrariamente un numero y, il problema richiede di verificare se $y \in S$. Possiamo affrontare la soluzione con due approcci differenti. Il primo è basato su una ricerca sequenziale dell'elemento y nella sequenza S; l'Algoritmo 9 fornisce una soluzione di complessità lineare $O(n)$.

Algoritmo 9 RICERCASEQUENZIALE($S = (x_1, x_2, \ldots, x_n)$, y)

Input: Una sequenza ordinata di numeri $S = (x_1, x_2, \ldots, x_n)$ e un numero y
Output: L'algoritmo stabilisce se $y \in S$

```
1: i := 1
2: fintanto che i ≤ n e x_i ≠ y ripeti
3:     i := i + 1
4: fine-ciclo
5: se i ≤ n allora
6:     restituisci "vero": y = x_i e quindi y ∈ S
7: altrimenti
8:     restituisci "falso": y ∉ S
9: fine-condizione
```

L'algoritmo di ricerca sequenziale non sfrutta in alcun modo l'ordinamento della sequenza S e dunque, esaminando uno ad uno gli elementi a partire dal primo, verifica se ciascun elemento coincide con il numero y cercato. Se $y = x_i$, per qualche $i = 1, 2, \ldots, n$, allora l'algoritmo impiegherà i operazioni (il ciclo 2–4 verrà iterato i volte) per portare a termine la ricerca; al contrario, se $y \notin S$, l'algoritmo dovrà eseguire tutti i confronti con ciascuno degli n elementi della sequenza, prima di poter stabilire che l'elemento non è presente; in questo caso, il "peggiore" per il nostro algoritmo, la complessità è $O(n)$.

Sfruttando però l'ordinamento crescente degli elementi di S è possibile attuare una strategia di *ricerca binaria* che ci consentirà di risparmiare parecchio tempo. Di fatto, a partire dall'elemento

collocato al centro della sequenza, $x_{[n/2]}$, si effettuerà la ricerca nella prima metà o nella seconda metà dell'insieme, in base all'esito del confronto tra y e tale elemento "mediano". Reiterando la stessa operazione anche sulla prima o sulla seconda metà dell'insieme, si suddividerà ancora a metà l'insieme entro cui eseguire la ricerca, fino ad individuare l'elemento y cercato, oppure concludere che tale elemento non è presente nella sequenza. L'Algoritmo 10 propone una pseudo-codifica di tale strategia.[5]

Algoritmo 10 RICERCABINARIA($S = (x_1, x_2, \ldots, x_n)$, y)

Input: Una sequenza ordinata di numeri $S = (x_1, x_2, \ldots, x_n)$ e un numero y
Output: L'algoritmo stabilisce se $y \in S$

```
1:  p := 1, r := n, q := ⌈(p+r)/2⌉
2:  fintanto che p < r e x_q ≠ y ripeti
3:      se y ≤ x_q allora
4:          r := q
5:      altrimenti
6:          p := q + 1
7:      fine-condizione
8:      q := ⌈(p+r)/2⌉
9:  fine-ciclo
10: se y = x_q allora
11:     restituisci "vero": y = x_q e quindi y ∈ S
12: altrimenti
13:     restituisci "falso": y ∉ S
14: fine-condizione
```

Con le variabili p e r si indicano rispettivamente gli indici degli estremi dell'intervallo di ricerca: dall'elemento x_p all'elemento x_r; pertanto inizialmente (passo 1) vengono assegnati i valori $p = 1$ e $r = n$; con la variabile q invece si tiene traccia dell'elemento centrale nella sequenza $x_p, \ldots, x_r$, ossia l'elemento il cui indice è dato dalla parte intera della media aritmetica tra p e r. La ricerca procede concentrandosi su sottoinsiemi di S sempre più piccoli fino ad individuare l'elemento cercato o, avendo ridotto la ricerca ad un sottoinsieme costituito da un solo elemento diverso da y, fino a concludere che il numero y non è presente nella sequenza S. Ad ogni iterazione del ciclo 2–9 il sottoinsieme entro cui si restringe la ricerca, viene ridotto della metà, per cui si giungerà rapidamente ad individuare l'elemento cercato o a concludere che tale elemento non esiste.

Aiutiamoci con un esempio a comprendere più a fondo la modalità con cui opera l'algoritmo. Supponiamo quindi che la sequenza S sia costituita da $n = 8$ elementi: $S = (x_1 = 15, x_2 = 22, x_3 = 36, x_4 = 47, x_5 = 59, x_6 = 87, x_7 = 92, x_8 = 99)$. Supponiamo di voler verificare se il numero $y = 60$ è presente nella sequenza S. Inizialmente l'intervallo di ricerca sarà costituito dall'intera sequenza S: $x_p = x_1 = 15$ e $x_r = x_8 = 99$; l'elemento collocato al centro della sequenza è l'elemento $x_4 = 47$ (si veda anche la schematizzazione riportata in Figura 2). Risulta $y > x_4$, per cui la ricerca si sposta sulla seconda metà dell'insieme, ridefinendo gli estremi dell'intervallo di ricerca: $x_p = x_5 = 59$, $x_r = x_8 = 99$ e dunque $x_q = x_6 = 87$. In questo caso risulta $y < x_6$, per cui l'intervallo si riduce della metà prendendo la prima parte della sottosequenza precedente: $x_p = x_5 = 59$, $x_r = x_6 = 87$, $x_q = x_5$. Risulta ancora una volta $y > x_5$, per cui dividendo a metà l'insieme si prende in esame la seconda

[5] Con la notazione $[n/2]$ si indica la parte intera del quoziente $n/2$.

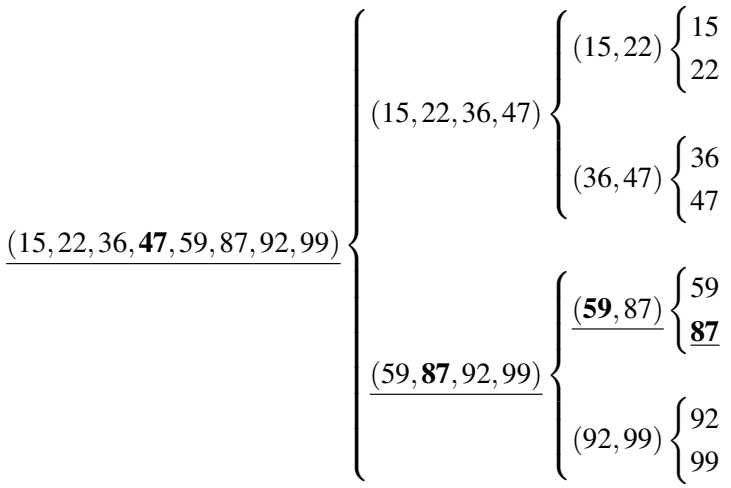

Figura 2: Esempio di suddivisione della sequenza per la ricerca binaria dell'elemento $y = 60$.

parte dell'ultima sottosequenza: $x_p = x_r = x_6 = 87$ e quindi anche $x_q = x_6$. Siccome $p = r$ il ciclo 2–9 termina: il confronto finale di y con x_q fallisce ($x_q \neq y$) e dunque si può concludere che $y \notin S$.

L'Algoritmo 10 esegue al massimo $\log_2 n$ iterazioni del ciclo principale costituito dai passi 2–9: infatti ad ogni iterazione l'intervallo di ricerca viene ridotto della metà e quindi, al massimo, potrà essere eseguito un numero di iterazioni pari alle suddivisioni a metà del numero intero n. Se $n = 2^k$ per qualche intero $k > 0$, allora una sequenza di n elementi può essere divisa a metà in due parti uguali esattamente k volte: $k = \log_2 n$. Di conseguenza, se $2^{k-1} < n < 2^k$ per qualche intero $k > 0$, lo stesso algoritmo non eseguirà più di $\log_2 n$ confronti, prima di individuare la soluzione. Quindi, sfruttando opportunamente l'ordinamento degli elementi della sequenza S, l'algoritmo si caratterizza con una complessità di $O(\log_2 n)$: la ricerca binaria è indubbiamente un metodo più efficiente della semplice ricerca sequenziale.

In questo modo abbiamo mostrato che esistono algoritmi di complessità sub-lineare o addirittura costante. Se in questi casi gli algoritmi caratterizzati da tale complessità sono indubbiamente molto efficienti, non sempre le cose vanno altrettanto bene: in alcuni casi, infatti, gli algoritmi hanno una complessità super-polinomiale, che li rende poco efficienti al punto da non essere considerati una strada percorribile nella pratica per giungere alla soluzione di un determinato problema.

Consideriamo ad esempio il seguente problema tipicamente combinatorio: dato un insieme $A = \{x_1, x_2, \ldots, x_n\}$ di numeri interi e fissata una soglia t, si chiede di stabilire se esiste un sottoinsieme S di A, tale che la somma dei suoi elementi sia esattamente t; in altri termini si vuole trovare, se esiste, $S \subseteq A$ tale che $\sum_{x_i \in S} x_i = t$. È un problema piuttosto semplice nei suoi termini generali e, nonostante questo, molto famoso e noto in letteratura con il nome di SUBSETSUM. Un algoritmo risolutore per questo problema opera costruendo tutti i sottoinsiemi di A e per ciascun sottoinsieme calcola la somma degli elementi; l'algoritmo termina quando ha individuato un sottoinsieme la cui somma sia proprio t, oppure quando, dopo aver calcolato tutti i possibili sottoinsiemi di A, termina concludendo che nessun sottoinsieme ha una somma pari a t. Non è un algoritmo particolarmente complicato, tuttavia, come verificheremo meglio nelle pagine del prossimo capitolo, siccome il numero di sottoinsiemi di un insieme A con n elementi è 2^n, la complessità dell'algoritmo nel caso peggiore è almeno pari al numero di sottoinsiemi che devono essere costruiti: $O(2^n)$.

$f(n)$	Dimensione del problema					
	$n = 10$	$n = 20$	$n = 40$	$n = 50$	$n = 100$	$n = 1.000$
$\log_2 n$	$3 \cdot 10^{-8}$ sec.	$4 \cdot 10^{-8}$ sec.	$5 \cdot 10^{-8}$ sec.	$5 \cdot 10^{-8}$ sec.	$6 \cdot 10^{-8}$ sec.	10^{-7} sec.
n	10^{-7} sec.	$2 \cdot 10^{-7}$ sec.	$4 \cdot 10^{-7}$ sec.	$5 \cdot 10^{-7}$ sec.	10^{-6} sec.	10^{-5} sec.
$n \log_2 n$	$3 \cdot 10^{-7}$ sec.	$8 \cdot 10^{-7}$ sec.	$2 \cdot 10^{-6}$ sec.	$2 \cdot 10^{-6}$ sec.	$6 \cdot 10^{-6}$ sec.	0,0001 sec.
n^2	10^{-6} sec.	$4 \cdot 10^{-6}$ sec.	0,000016 sec.	0,000025 sec.	0,0001 sec.	0,01 sec.
n^3	0,00001 sec.	0,00008 sec.	0,00064 sec.	0,00125 sec.	0,01 sec.	10 sec.
n^4	0,0001 sec.	0,0016 sec.	0,0256 sec.	0,0625 sec.	1 sec.	16 min.
2^n	0,00001 sec.	0,01 sec.	3 ore	130 gg	miliardi di anni	...
$n!$	0,036 sec.	7 secoli	miliardi di anni	...	...	...

Tabella 1: Tempo di esecuzione di algoritmi con funzioni di complessità differenti al variare della dimensione del problema su un calcolatore che esegue cento milioni di operazioni al secondo.

Proviamo a "visualizzare" con un esempio concreto quanto detto fino ad ora, per comprendere la reale portata di quanto abbiamo affermato. Supponiamo quindi di disporre di un computer di media potenza, in grado di eseguire cento milioni di operazioni elementari al secondo; in altri termini supponiamo che la macchina di calcolo di cui disponiamo impieghi 10^{-8} secondi per eseguire una singola operazione. Non si tratta di un super-calcolatore: una workstation abbastanza potente oggi ha delle performance analoghe.

Supponiamo inoltre di disporre di un insieme di algoritmi caratterizzati da complessità computazionali differenti e di eseguire tali algoritmi sul nostro calcolatore, per risolvere problemi di dimensione via via sempre più grande: prima una semplice istanza con soli 10 elementi da elaborare, poi un'istanza con 20 elementi, fino ad arrivare ad istanze del problema con 100 o 1.000 elementi. Calcoliamo quindi il tempo necessario per eseguire i diversi algoritmi per risolvere tali istanze del problema. In Tabella 1 sono riportati, con una certa approssimazione per difetto, i tempi necessari a risolvere ciascuna istanza del problema utilizzando i diversi algoritmi; ricordiamo comunque che la macchina su cui vengono eseguiti è sempre la stessa, la nostra workstation da 100 milioni di operazioni al secondo.

Osservando la tabella possiamo notare come gli algoritmi caratterizzati da funzioni di complessità lineari, logaritimiche e polinomiali (del tipo n^k, per qualche costante $k > 1$ fissata) presentano dei tempi di calcolo che crescono in modo accettabile al crescere delle dimensioni del problema. Attendere qualche secondo o addirittura qualche minuto per ottenere il risultato di un problema calcolato su un'istanza di grandi dimensioni è sicuramente accettabile in molti contesti applicativi. Ad esempio un buon algoritmo di ordinamento, come il *merge sort*, caratterizzato da una complessità $O(n \log_2 n)$ è in grado di ordinare un archivio di un milione di record in meno di un secondo.

Ben diversa è invece la situazione nel caso in cui si adotti un algoritmo di complessità *super polinomiale*, ossia espressa da una funzione del tipo k^n, per qualche valore costante $k > 1$ fissato. Come si vede chiaramente dai risultati esposti in Tabella 1, anche su istanze del problema di piccola entità ($n = 20, 40, 50$) il tempo richiesto da tali algoritmi per calcolare la soluzione diventa immediatamente inaccettabile. D'altra parte in questi casi non è pensabile neanche aggirare il problema acquistando un calcolatore 100 volte più potente, perché comunque anche in tal caso sarà impossibile affrontare il calcolo della soluzione del problema per istanze anche poco più grandi. Analoga considerazione è possibile fare ipotizzando di affrontare il problema mediante un apparato di calcolo parallelo, che affidi la ricerca della soluzione a più macchine, che operino contemporaneamente nella ricerca della soluzione. Anche in questo caso, pur dividendo i tempi di calcolo per due, dieci o cento, non si otterrebbe comunque un miglioramento significativo per istanze del problema di dimensioni non banali (es.: $n = 100$ o $n = 1.000$).

La crescita del tempo necessario per calcolare la soluzione del problema con algoritmi di complessità super-polinomiale è infatti troppo rapida rispetto alla crescita della dimensione dell'istanza del problema: utilizzando un algoritmo di complessità $O(2^n)$, aumentando di un solo elemento la dimensione dell'istanza del problema, il tempo di calcolo necessario viene raddoppiato; con un algoritmo di complessità $O(n!)$ il tempo di calcolo necessario per calcolare la soluzione di un'istanza di dimensione n aumenta di n volte rispetto al tempo necessario per calcolare la soluzione di un'istanza del problema di dimensione $n-1$.

Il calcolo della complessità di un algoritmo è un fatto tutt'altro che semplice; per capire meglio il concetto è possibile comunque partire da alcuni aspetti essenziali di facile comprensione. Nel calcolo della complessità infatti è opportuno distinguere il contributo delle diverse componenti di uno stesso algoritmo: se infatti un determinato procedimento di calcolo è costituito dalla concatenazione di due o più procedure, distinte l'una dall'altra, allora la complessità dell'intero algoritmo sarà data dalla somma delle complessità di ciascuna componente. Una determinata procedura ha una complessità costante se si limita ad eseguire un numero fissato di operazioni, del tutto indipendente dalla dimensione n del problema. La complessità di una procedura cresce in presenza di cicli che consentono di iterare più volte un determinato blocco di istruzioni; in questi casi la complessità può essere calcolata moltiplicando il numero di iterazioni del ciclo per la complessità del blocco di istruzioni iterate dal ciclo stesso. Tipicamente quindi un termine lineare in n viene introdotto da una procedura che itera per n volte un insieme costante di istruzioni; termini polinomiali di grado superiore generalmente sono dovuti alla presenza di cicli nidificati l'uno nell'altro. I termini logaritmici, infine, come abbiamo visto nelle pagine precedenti, sono dovuti a strutture algoritmiche iterative che sono in grado di scartare molti elementi evitando un'elaborazione "a tappeto" che richiederebbe un tempo di esecuzione almeno lineare in n.

6 Classi di complessità per i problemi

Calcolandone la complessità computazionale, è possibile quindi stabilire il grado di efficienza di un algoritmo: tanto più basso è il numero di operazioni eseguite per calcolare la soluzione di un problema, tanto maggiore sarà l'efficienza dell'algoritmo. In questo modo è possibile confrontare fra loro algoritmi differenti che consentono di risolvere il medesimo problema, determinando con certezza quale sia la strategia risolutiva più efficiente, con cui si giunge al calcolo della soluzione nel minor tempo possibile.

È utile estendere il concetto di complessità dagli algoritmi ai problemi: è possibile identificare la complessità computazionale che caratterizza intrinsecamente un determinato problema, a prescindere dalla complessità di uno specifico algoritmo risolutivo per il problema stesso? È una questione importante posta dagli studiosi della teoria della calcolabilità e della complessità che ha trovato risposta nella seguente affermazione: definiamo come complessità computazionale di un problema la complessità dell'algoritmo più efficiente in grado di risolvere tale problema; non è necessario esibire un simile algoritmo, è sufficiente dimostrare che un tale algoritmo esiste e che tale è la sua complessità computazionale. Una volta definita la complessità di un problema, diremo che ogni algoritmo caratterizzato dalla medesima complessità è un algoritmo *ottimo* per la soluzione di tale problema.

Ad esempio esistono numerosi algoritmi per la risoluzione del problema dell'ordinamento di una sequenza di n elementi: si va dagli algoritmi elementari come *selection sort* ed *insertion sort* la cui complessità è $O(n^2)$, ad algoritmi più sofisticati ed efficienti, come *merge sort* e *heap sort* caratterizzati da una complessità $O(n\log_2 n)$. Tutti questi algoritmi sono in grado di risolvere il problema dell'ordinamento di una sequenza, ma lo fanno con strategie differenti, che determinano configura-

zioni più o meno favorevoli per l'algoritmo.[6] Qual è dunque la complessità del problema dell'ordinamento? È una richiesta di cruciale importanza, dal momento che dalla risposta che daremo a questo quesito potremo stabilire se esistono dei margini di miglioramento nella ricerca di algoritmi ancor più efficienti, o se invece siamo già in possesso di algoritmi *ottimi* per la soluzione del problema in esame.

Il seguente Teorema afferma che qualunque algoritmo che affronta il problema dell'ordinamento di una sequenza, senza nessuna conoscenza specifica sull'insieme da ordinare e basando la propria strategia risolutiva solo su una successione di confronti tra gli elementi dell'insieme, possiede una complessità non inferiore a $O(n\log_2 n)$: utilizzando la notazione Ω introdotta a pagina 16, possiamo dire che la complessità di tali algoritmi è $\Omega(n\log_2 n)$. Pertanto questa è la complessità computazionale del problema dell'ordinamento. Possiamo così concludere che gli algoritmi *merge sort* e *heap sort*, che hanno proprio la complessità $O(n\log_2 n)$, sono algoritmi *ottimi* per il problema dell'ordinamento.

Teorema 1. *La complessità di un algoritmo per la soluzione del problema dell'ordinamento di una sequenza di n elementi, attraverso le operazioni di confronto, è* $\Omega(n\log_2 n)$.

Dimostrazione. Possiamo schematizzare il funzionamento di un qualsiasi algoritmo di ordinamento per confronto, attraverso un "albero di decisioni": ad ogni passo dell'algoritmo vengono confrontati fra loro due elementi e, a seconda dell'esito del confronto, viene seguita una delle due possibili strade conseguenti, confrontando fra loro altri due elementi della sequenza. Il procedimento termina giungendo ad una permutazione degli elementi della sequenza iniziale. Tali permutazioni sono le foglie, i nodi terminali, dell'albero di decisione. Siccome le possibili permutazioni degli elementi di un insieme di cardinalità n è pari ad $n!$, ne segue che le foglie dell'albero sono almeno $n!$: infatti due o più successioni distinte di confronti potrebbero portare alla medesima permutazione finale.

Il numero di operazioni effettuate dall'algoritmo di ordinamento è dato dalla profondità dell'albero di decisione; tale albero è un albero binario (ogni vertice ha al massimo due figli) per cui, se la sua profondità è h l'albero non ha più di 2^h foglie. Per quanto detto in precedenza si ha quindi $n! \leq 2^h$, per cui, ricorrendo anche all'approssimazione di Stirling del fattoriale[7], si ottiene $h \geq \log_2 n! \cong \Omega(n\log_2 n)$. □

Dunque possiamo associare ogni problema ad una specifica classe di complessità computazionale; in effetti, ad essere più precisi possiamo fare questo per qualunque problema per cui si conosca un algoritmo risolutore, oppure per quei problemi per i quali è stato dimostrato che un algoritmo risolutore esiste. Tuttavia non tutti i problemi rientrano in questa grande famiglia: esiste infatti anche una famiglia di problemi che non sono risolubili mediante un procedimento di calcolo algoritmico, i problemi che non sono Turing-calcolabili a cui abbiamo fatto cenno a pagina 14 parlando di problemi *indecidibili*.

Ferma restando l'esistenza della classe dei problemi non calcolabili (o indecidibili), d'ora in avanti ci occuperemo soltanto di problemi la cui soluzione possa essere calcolata mediante un procedimento algoritmico. Dunque, nel seguito di questo capitolo, intendiamo proporre un modello per raggruppare i problemi in base al loro maggiore o minore "grado di difficoltà": così come abbiamo fatto nella sezione precedente misurando l'efficienza degli algoritmi, intendiamo ora misurare e mettere a confronto la difficoltà difficoltà di problemi differenti. Di fatto è questa l'essenza della teoria della calcolabilità e della complessità computazionale che coinvolge da alcune decine di anni i migliori esperti di informatica teorica in tutto il mondo.

[6] Per una disamina dettagliata di questi algoritmi è possibile fare riferimento ad alcuni dei testi citati in bibliografia, come ad esempio [2] e [3].

[7] Per valori di n sufficientemente grandi, è possibile approssimare la funzione $n!$ con la seguente formula di Stirling: $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1+\Theta(1/n))$.

Innanzi tutto, per poter confrontare fra loro problemi differenti, dobbiamo formularli in modo omogeneo. Per far questo distinguiamo le seguenti grandi famiglie di problemi:

Problemi di decisione sono formulati in modo tale da richiedere una risposta di tipo "binario": *sì* o *no*, *vero* o *falso*, ecc. Sono anche detti *problemi di esistenza*, perché senza che ci sia bisogno di esibire una soluzione, è sufficiente stabilire se tale soluzione esiste. Ad esempio: «esiste un soluzione per l'equazione $f(x) = 0$ nell'intervallo $[a,b]$?» Al fine di poter confrontare facilmente fra loro i problemi si usa formularli in versione decisionale.

Problemi di ricerca sono problemi che richiedono di esibire una possibile soluzione del problema (se esiste) e non solo di valutarne l'esistenza. Ad esempio: «esibire, se esiste, una soluzione per l'equazione $f(x) = 0$ nell'intervallo $[a,b]$».

Problemi di enumerazione in questo caso si richiede di calcolare tutte le soluzioni del problema. Ad esempio: «calcolare tutte le soluzioni per l'equazione $f(x) = 0$ nell'intervallo $[a,b]$».

Problemi di ottimizzazione in quest'ultima famiglia rientrano quei problemi che chiedono di calcolare la soluzione ottimale per l'istanza in esame; non soltanto è necessario stabilire se esiste almeno una soluzione o, in caso affermativo, di produrne una qualsiasi o tutte le soluzioni possibili, indistintamente: in questo caso è richiesto di esibire la migliore soluzione, quella che rende minimo o massimo (a seconda dei casi) un determinato criterio di valutazione, detto *funzione obiettivo*. Ad esempio: «calcolare una soluzione $\bar{x}$ per l'equazione $f(x) = 0$ nell'intervallo $[a,b]$, tale da rendere minimo il valore della funzione $\varphi(x)$».

Definiamo la classe di complessità P dei **problemi polinomiali** come l'insieme di tutti i problemi che ammettono un algoritmo risolutivo di complessità polinomiale. Ad esempio il problema dell'ordinamento di una sequenza o il problema della ricerca di un elemento in un insieme sono entrambi membri della classe P, visto che, come abbiamo osservato nelle pagine precedenti, ammettono entrambi un algoritmo risolutore di complessità $O(n^2)$.

Diremo che i problemi che, pur rientrando nella famiglia dei problemi calcolabili con un procedimento algoritmico, non ammettono un algoritmo risolutore di complessità polinomiale, o per i quali non conosciamo ancora un algoritmo di questo genere, sono **problemi intrattabili**. È chiaro che questo tipo di problemi rivestono un particolare interesse per gli studiosi della teoria degli algoritmi e della calcolabilità; peraltro molti dei problemi di ottimizzazione combinatoria rientrano in questa famiglia. Pertanto non possiamo accontentarci di classificare semplicemente come *intrattabili* una classe così ampia di problemi.

Definiamo quindi la classe di complessità NP (dall'inglese: *Nondeterministic Polynomial-time*). Non è la classe dei problemi "non polinomiali", come si potrebbe desumere ingenuamente dal nome NP; è l'insieme dei problemi che possono essere risolti in tempo polinomiale da un **algoritmo non deterministico**[8].

In un algoritmo *deterministico* è sempre univocamente determinato quale sia il valore di ciascuna variabile in uno specifico momento di esecuzione dell'algoritmo; siccome il valore delle variabili in un certo istante è univocamente determinato dalla struttura dell'algoritmo e dalla istanza del problema che si sta risolvendo, è anche determinato univocamente quale sia la prossima operazione che sarà eseguita dall'algoritmo stesso. In un algoritmo *non deterministico*, invece, ammettiamo l'esistenza di un'istruzione speciale, denominata "**choice**", che permette all'algoritmo di assegnare ad una stessa variabile tutti i valori presenti in un determinato insieme e di proseguire l'esecuzione dell'algoritmo

[8] Si veda a questo proposito [12].

con tante istanze parallele, quanti sono i possibili valori assegnati alla variabile. Se X è un insieme finito, con $|X| = k$, con l'istruzione «$x := \text{choice}(X)$» saranno assegnati k valori diversi a k istanze diverse della variabile x; quindi l'algoritmo proseguirà la sua esecuzione su "k percorsi paralleli", uno per ciascun valore assegnato ad x.

Quando una delle istanze dell'algoritmo eseguite in parallelo termina, lo comunica a tutte le altre indicando se ha raggiunto una soluzione del problema o se ha terminato l'esecuzione senza trovarla: nel primo caso si utilizza l'istruzione "**success**", mentre nel secondo caso si utilizza "**failure**". Se uno degli algoritmi termina con l'istruzione "success", allora anche tutte le altre istanze eseguite in parallelo termineranno immediatamente, interrompendo la ricerca della soluzione; viceversa, se un'istanza termina con l'istruzione "failure", le altre istanze proseguiranno l'esecuzione, nella speranza di individuare una soluzione dell'istanza del problema.

Un algoritmo di questo genere, in generale, non è possibile realizzarlo all'atto pratico, pur utilizzando un computer con capacità di calcolo parallelo dotato di numerose CPU, in quanto il numero di processi paralleli che in teoria deve poter essere eseguito da un algoritmo non deterministico, è infinito o comunque potenzialmente superiore alle capacità di parallelismo di qualsiasi computer realizzabile nella pratica. Ciò nonostante il concetto di algoritmo non deterministico è uno strumento molto potente utilizzato nello studio della teoria della calcolabilità.

A titolo di esempio riportiamo alcuni algoritmi non deterministici per il calcolo della soluzione di problemi ben noti. Nel primo caso (Algoritmo 11) consideriamo il problema dell'ordinamento: l'istanza del problema è costituita da una sequenza $A = \langle a_1, \ldots, a_n \rangle$ di elementi (ad esempio numeri interi) su cui è definita una relazione d'ordine. Si vuole calcolare una permutazione dell'insieme, tale che $a_i \leq a_{i+1}$ per ogni $i = 1, 2, \ldots, n-1$. Conosciamo numerosi algoritmi deterministici per la soluzione di questo problema, ma, a titolo di esempio, riportiamo un possibile algoritmo non deterministico.

Algoritmo 11 ORDINAMENTONONDETERMINISTICO($A = \langle a_1, \ldots, a_n \rangle$)

Input: Una sequenza di numeri $A = \langle a_1, \ldots, a_n \rangle$
Output: Una permutazione di A con gli elementi disposti in ordine crescente

```
1:  i := choice(⟨1,...,n⟩)
2:  scambia(a_1, a_i)
3:  per i := 2,...,n ripeti
4:      j := choice(⟨i,...,n⟩)
5:      scambia(a_i, a_j)
6:      se a_i < a_{i-1} allora
7:          failure
8:      fine-condizione
9:  fine-ciclo
10: success
```

Al passo 1 viene selezionato l'indice dell'elemento da collocare come primo elemento della permutazione: esistono n possibili scelte e dunque n possibili valori per la variabile i: dunque l'istruzione riportata al passo 2 e tutte le istruzioni successive, saranno eseguite "in parallelo" da n diverse istanze dell'algoritmo non deterministico; la prima istanza avrà come valore $i = 1$, la seconda $i = 2$ e così via fino all'n-esima istanza che avrà come valore $i = n$.

Lo stesso meccanismo si sviluppa al passo 4 dell'algoritmo, quando alla variabile j vengono assegnati numerosi valori distinti, selezionati nell'insieme $\{i, i+1, i+2, \ldots, n\}$; in seguito all'esecuzione dell'istruzione "choice" riportata al passo 4, l'algoritmo si suddividerà in $n - i + 1$ istanze parallele, ciascuna con un valore differente della variabile j.

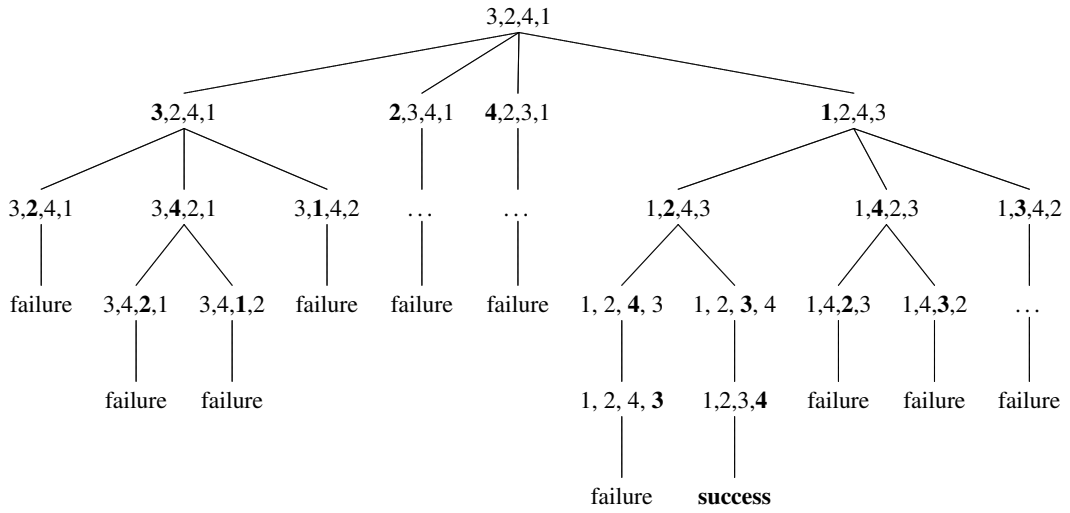

Figura 3: Un esempio di albero delle operazioni svolte dall'algoritmo non deterministico ORDINAMENTO-NONDETERMINISTICO per risolvere l'istanza del problema $\langle 3,2,4,1 \rangle$.

In questo modo, le diverse strade che vengono percorse dall'algoritmo, ciascuna con differenti assegnazioni di valori alle variabili, possono essere rappresentate come un albero, la cui radice si trova nella prima istruzione dell'algoritmo e in cui ciascun vertice rappresenta un'operazione elementare eseguita dall'algoritmo stesso; i vertici con due o più figli rappresentano le istruzioni "choice", in cui l'algoritmo si suddivide in due o più istanze parallele. Le foglie dell'albero rappresentano le istruzioni di terminazione "success" o "failure" (vedi Figura 3). Il cammino più breve tra la radice dell'albero e una delle foglie di tipo "success" rappresenta la sequenza di operazioni e di "scelte" che devono essere eseguite per risolvere il problema nel tempo più breve, ossia, la sequenza di istruzioni di una delle istanze parallele dell'algoritmo non deterministico, che definisce il tempo di esecuzione dell'algoritmo stesso. Dunque se la lunghezza di tale cammino è "polinomiale" nella dimensione dell'istanza del problema, potremo concludere che l'algoritmo non deterministico risolve il problema in tempo polinomiale.

Proponiamo un algoritmo non deterministico per la soluzione del problema della ricerca del cammino di lunghezza minima tra due vertici u e v di un grafo G (Algoritmo 12). Questa volta l'algoritmo è di tipo "ricorsivo", ossia sono presenti delle istruzioni in cui l'algoritmo richiama se stesso (riga 9).

Anche in questo caso l'algoritmo si suddivide in tante istanze parallele quanti sono i vertici adiacenti al vertice u, con l'istruzione a riga 1 (con $N(u)$ indichiamo l'insieme dei vertici adiacenti a u). Dunque l'esecuzione dell'algoritmo per la soluzione di un'istanza del problema potrà essere rappresentata da un albero in cui i vertici con due o più figli rappresentano le istruzioni "choice".

Il cammino dalla sorgente ad una delle foglie rappresenta una successione di scelte che porta a determinare che una determinata istanza del problema possiede o meno una soluzione: se tutte le foglie sono di tipo "failure" allora l'istanza del problema non ammette alcuna soluzione (es.: nel caso del cammino minimo tra due vertici di un grafo, se il grafo non è connesso e i due vertici appartengono a due componenti distinte, il cammino non esiste). Se invece esiste almeno una foglia di tipo "success", il cammino dalla radice ad una di quelle foglie rappresenta la sequenza di scelte che devono essere compiute per giungere a tale soluzione. In altri termini, quei cammini rappresentano anche le operazioni che devono essere eseguite per verificare la correttezza di una determinata soluzione di un problema.

Algoritmo 12 CAMMINOMINIMONONDETERMINISTICO(G,u,v)

Input: Un grafo $G=(V,E)$ e due vertici $u,v \in V(G)$
Output: Il cammino di lunghezza minima su G da u a v

```
1: x := choice(N(u))
2: se x = v allora
3:     success
4: altrimenti
5:     se colore(x) = grigio oppure x = u allora
6:         failure
7:     altrimenti
8:         colore(x) := grigio
9:         CAMMINOMINIMONONDETERMINISTICO(G,x,v)
10:    fine-condizione
11: fine-condizione
```

Possiamo quindi fornire una definizione alternativa della classe NP, come l'insieme dei problemi che ammettono un **algoritmo di verifica** di complessità polinomiale. Un algoritmo di questo genere è tale da poter stabilire se una soluzione "presunta" del problema in esame sia una soluzione "effettiva" oppure no; se tale algoritmo è caratterizzato da una complessità polinomiale, allora il problema è NP. Ad esempio il problema dell'ordinamento è NP, dal momento che mediante l'Algoritmo 8, di complessità polinomiale, è possibile verificare se una permutazione degli elementi dell'insieme da ordinare, fornita come "soluzione presunta", sia effettivamente una soluzione del problema (ossia una permutazione degli elementi tale da rispettare l'ordine non decrescente) oppure no.

Dalla definizione delle due precedenti classi di complessità risulta che P $\subseteq$ NP; infatti se esiste un algoritmo polinomiale per *calcolare* la soluzione del problema, allora lo stesso algoritmo può essere utilizzato anche per *verificare* una "soluzione presunta", semplicemente calcolando la soluzione esatta e poi verificando che le due soluzioni coincidano o che abbiano la medesima caratteristica.

Il problema SUBSETSUM presentato nelle pagine precedenti è un problema NP: dato un sottoinsieme S dell'insieme A è possibile verificare in tempo lineare se la somma degli elementi di S è uguale alla soglia t oppure no. Dunque il problema ammette un algoritmo di verifica di complessità polinomiale. Tuttavia, come abbiamo accennato in precedenza, l'algoritmo di enumerazione dei sottoinsiemi di A, per una ricerca "esaustiva" di un eventuale sottoinsieme di somma t, ha una complessità superpolinomiale; al momento non si conosce nessun algoritmo in grado di risolvere SUBSETSUM in tempo polinomiale.

Dati due problemi $\mathcal{A}$ e $\mathcal{B}$ possiamo *ridurre* il problema $\mathcal{A}$ al problema $\mathcal{B}$ se riusciamo a trasformare ogni istanza α di $\mathcal{A}$ in una istanza β di $\mathcal{B}$, in modo tale che la soluzione di β sia anche la soluzione di α. Se questa trasformazione avviene mediante un **algoritmo di riduzione** di complessità polinomiale, allora diremo che $\mathcal{A}$ è **riducibile in tempo polinomiale** a $\mathcal{B}$ e scriveremo $\mathcal{A} \leq_P \mathcal{B}$. Se vale tale relazione allora possiamo affermare che il problema $\mathcal{A}$ non è più difficile del problema $\mathcal{B}$.

La riducibilità in tempo polinomiale di un problema ad un altro gode della proprietà transitiva, come viene affermato dalla seguente Proposizione.

Proposizione 2. *Siano $\mathcal{A}$, $\mathcal{B}$ e $\mathcal{C}$ tre problemi, tali che $\mathcal{A} \leq_P \mathcal{B}$ e $\mathcal{B} \leq_P \mathcal{C}$. Allora risulta $\mathcal{A} \leq_P \mathcal{C}$.*

Dimostrazione. Se $\mathcal{A} \leq_P \mathcal{B}$ allora esiste un algoritmo di riduzione $\mathcal{R}_1$ di complessità polinomiale; inoltre se $\mathcal{B} \leq_P \mathcal{C}$, allora esiste un secondo algoritmo di riduzione $\mathcal{R}_2$ di complessità polinomiale.

Dunque, concatenando il primo ed il secondo algoritmo di riduzione si ottiene un algoritmo di riduzione $\mathcal{R}$ che trasforma le istanze di $\mathcal{A}$ in istanze di $\mathcal{C}$; la complessità di tale algoritmo è data al più dalla somma della complessità di $\mathcal{R}_1$ e $\mathcal{R}_2$ ed è pertanto anch'esso di complessità polinomiale. □

Visto che la concatenazione di algoritmi di complessità polinomiale, dà luogo ad un algoritmo di complessità polinomiale, possiamo dimostrare facilmente la seguente importante proprietà.

Teorema 2. *Siano $\mathcal{A}$ e $\mathcal{B}$ due problemi tali che $\mathcal{A} \leq_P \mathcal{B}$. Allora $\mathcal{B} \in P \implies \mathcal{A} \in P$.*

Dimostrazione. La composizione in sequenza di due algoritmi polinomiale dà luogo ad un algoritmo polinomiale: dunque componendo l'algoritmo di riduzione polinomiale di $\mathcal{A}$ in $\mathcal{B}$ e l'algoritmo risolutivo di $\mathcal{B}$, si ottiene un algoritmo risolutivo per il problema $\mathcal{A}$ di complessità polinomiale. □

Consideriamo, a titolo di esempio, due problemi NP molto noti e la cui formulazione è assai semplice e proviamo che sono riducibili in tempo polinomiale l'uno all'altro:

1. SUBSETSUM: Dato un insieme finito $A \subset \mathbb{N}$ e un intero $b > 0$, esiste un sottoinsieme $A' \subseteq A$ tale che $\sum_{a \in A'} a = b$?
2. PARTITION: Dato un insieme finito $A \subset \mathbb{N}$ esiste un sottoinsieme $A' \subseteq A$ tale che $\sum_{a \in A'} a = \sum_{a \in A \setminus A'} a$?

Entrambi i problemi sono NP, perché in entrambi i casi per verificare che una determinata soluzione (un sottoinsieme di A) sia ammissibile per una determinata istanza del problema, basta effettuare la somma degli elementi di A' o di A. Dunque la verifica della soluzione può essere effettuata in tempo polinomiale.

Proposizione 3. *Risulta* PARTITION $\leq_P$ SUBSETSUM.

Dimostrazione. Vogliamo dimostrare che ogni istanza di PARTITION può essere trasformata in tempo polinomiale in un'istanza equivalente di SUBSETSUM e che la soluzione di questa istanza è anche soluzione dell'istanza di PARTITION ad essa associata.

Sia $A \subset \mathbb{N}$ un'istanza di PARTITION. Sia $n = |A|$ e $s = \sum_{i=1}^{n} a_i$. Il problema PARTITION richiede di trovare un sottoinsieme di A tale che la somma degli elementi in A' sia pari alla somma degli elementi di A che non sono in A'. Dunque si cerca un insieme A' la cui somma degli elementi sia pari a $s/2$:

$$\sum_{a \in A'} a = \sum_{a \in A \setminus A'} a \implies \sum_{a \in A'} a = \frac{1}{2} \sum_{a \in A}$$

Quindi ogni istanza di PARTITION può essere trasformata nell'istanza equivalente di SUBSETSUM data da $(A, s/2)$. Se esiste un sottoinsieme $A' \subseteq A$ che risolve il problema SUBSETSUM con la soglia $s/2$ allora, tale soluzione risolve anche l'istanza associata del problema PARTITION. □

Proposizione 4. *Risulta* SUBSETSUM $\leq_P$ PARTITION.

Dimostrazione. Consideriamo l'istanza di SUBSETSUM data dalla coppia $(A = \{a_1, \ldots, a_n\}, k)$. Sia $s = \sum_{i=1}^{n} a_i$. Possiamo allora costruire un'istanza di PARTITION sull'insieme $A^+ = \{a_1, \ldots, a_n, b, c\}$, ponendo $b = 2s - k$ e $c = s + k$.

Infatti la somma degli elementi dell'insieme A^+ è $s + b + c = s + 2s - k + s + k = 4s$. Per cui per risolvere tale istanza del problema PARTITION dobbiamo trovare una partizione di A^+ di somma $2s$.

Ma è facile osservare che esiste una soluzione di PARTITION, un sottoinsieme di $\{a_1,\ldots,a_n,b,c\}$ di somma $2s$, se e solo se esiste un sottoinsieme di $\{a_1,\ldots,a_n\}$ di somma k, ossia una soluzione di SUBSETSUM/ Infatti, riordinando per semplicità gli elementi $a_1,\ldots,a_n$ possiamo scrivere:

$$\overbrace{\overbrace{a_1+\cdots+a_i}^{k}+\overbrace{a_{i+1}+\cdots+a_n}^{s-k}}^{s}+\overbrace{\overbrace{b}^{2s-k}+\overbrace{c}^{s+k}}^{3s}=$$
$$=\underbrace{\left(\underbrace{b}_{2s-k}+\underbrace{a_1+\cdots+a_i}_{k}\right)}_{2s}+\underbrace{\left(\underbrace{a_{i+1}+\cdots+a_n}_{s-k}+\underbrace{c}_{s+k}\right)}_{2s}$$

□

Per spingere in avanti la ricerca nel campo dei problemi apparentemente intrattabili, nel 1971 Cook propose la classe NPC dei **problemi NP-completi** definita come segue: un problema $\mathcal{A}$ appartiene alla classe NPC se soddisfa entrambi i seguenti requisiti:

1. $\mathcal{A} \in$ NP;
2. $\mathcal{B} \leq_P \mathcal{A}$ per ogni $\mathcal{B} \in$ NP.

Nel 1971, Cook dimostrò formalmente che la classe NPC non è vuota e contiene almeno un elemento. Per poter formulare questo importante risultato dobbiamo prima definire il problema SAT (*satisfiability*): dato un insieme U di variabili booleane e un'espressione logica formulata con un insieme C di clausole su U, esiste un'assegnazione di valori alle variabili di U tale da rendere vera l'espressione?

Ad esempio, consideriamo la seguente espressione formata utilizzando le variabili dell'insieme $U = \{x_1, x_2, x_3\}$:

$$((x_1 \wedge x_2) \vee (x_1 \wedge x_3)) \wedge (x_2 \iff \neg x_3)$$

L'espressione risulta vera per $x_1 =$ vero, $x_2 =$ vero, $x_3 =$ falso e per $x_1 =$ vero, $x_2 =$ falso, $x_3 =$ vero, mentre risulta falsa per ogni altra assegnazione di valori alle variabili di U. Al contrario, l'espressione $x_1 \wedge x_2) \wedge (x_1 \wedge \neg x_2)$ non ammette alcuna assegnazione di valori alle variabili booleane x_1 e x_2 tale da rendere vera l'intera espressione.

Teorema 3 (Cook – 1971)**.** *SAT è un problema NP-completo.*

Questo risultato è di importanza fondamentale per la teoria degli algoritmi e per lo studio della calcolabilità, infatti, a partire dal Teorema di Cook, per provare che un problema $\mathcal{A}$ è NP-completo, basta mostrare che esiste un solo problema $\mathcal{B}$ NP-completo che sia riducibile ad $\mathcal{A}$ in tempo polinomiale. Infatti, se $\mathcal{B} \in$ NPC allora ogni problema $\mathcal{C} \in$ NP è tale che $\mathcal{C} \leq_P \mathcal{B}$ e quindi, per la transitività (Proposizione 2), risulta anche che $\mathcal{C} \leq_P \mathcal{A}$.

In generale, quindi, per dimostrare che $\mathcal{A}$ è un problema NP-completo, bisogna dimostrare che $\mathcal{A} \in$ NP e, successivamente, scegliere un qualsiasi problema $\mathcal{B} \in$ NPC e dimostrare che $\mathcal{B} \leq_P \mathcal{A}$.

Ad esempio si dimostra che PARTITION e SUBSETSUM sono entrambi problemi NP-completi, attraverso la seguente catena di riduzioni: SAT $\leq_P$ 3SAT $\leq_P$ 3DM $\leq_P$ PARTITION $\leq_P$ SUBSETSUM.

Il problema 3DM (*three-dimensional matching*) è definito come segue: dati tre insiemi finiti X, Y e Z, possiamo definire l'insieme $T = X \times Y \times Z$; fissato $k > 0$, esiste un sottoinsieme $M \subset T$ di

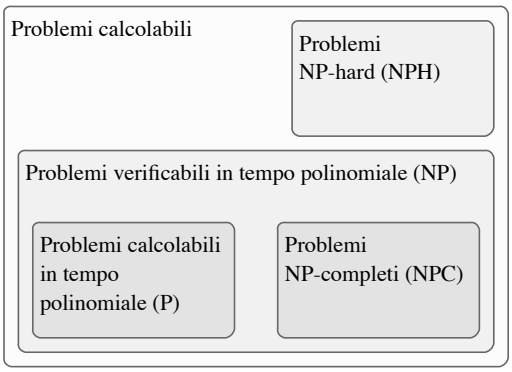

Figura 4: Diagramma delle classi di complessità P, NP, NPC e NPH.

cardinalità $|M| > k$ tale che gli elementi di M abbiano a due a due nessun elemento della terna in comune? Ossia, per ogni $a,b \in M$ deve risultare $a = (x_1,y_1,z_1), b = (x_2,y_2,z_2), x_1 \neq x_2, y_1 \neq y_2, z_1 \neq z_2$.

È possibile definire un altro problema, analogo a 3DM, denominato *stable marriage problem* (problema del matrimonio stabile – STABLEMARRIAGE): dati due insiemi finiti X e Y e un sottoinsieme M di $X \times Y$, esiste un sottoinsieme di M tale che le coppie in M non abbiano elementi in comune? Ossia, esiste un sottoinsieme $M' \subset M$ tale che per ogni $(x_1,y_1),(x_2,y_2) \in M'$ risulta $x_1 \neq x_2$, $y_1 \neq y_2$?

È interessante osservare che STABLEMARRIAGE è un problema di complessità polinomiale, ossia, come vedremo, esiste un algoritmo di complessità polinomiale che lo risolve, mentre $3DM$ è NP-completo e, ad oggi, nessuno ha dimostrato che esiste un algoritmo di complessità polinomiale in grado di risolverlo.

In modo informale possiamo affermare che i problemi NPC sono, tra i problemi NP, quelli più difficili, ai quali si può ridurre ogni altro problema NP. La definizione della classe NPC ne mette in evidenza la proprietà più interessante che possiamo formulare nel seguente Teorema.

Teorema 4. *Sia $\mathcal{A}$ un qualsiasi problema NP-completo. Se $\mathcal{A} \in P$ allora $P = NP$.*

Dimostrazione. Se venisse individuato un algoritmo risolutivo di complessità polinomiale per il problema $\mathcal{A}$, allora per il Teorema 2 risulterebbe polinomiale ogni problema riducibile ad $\mathcal{A}$. Dunque, per la definizione della classe NPC e visto che per ipotesi $\mathcal{A} \in$ NPC, ogni problema NP risulterebbe polinomiale; dunque NP $\subseteq$ P. Siccome risulta anche P $\subseteq$ NP allora questo implica che P $=$ NP. □

Stabilire se P $=$ NP è il più importante problema della teoria della calcolabilità e di informatica teorica tutt'ora irrisolto. Il precedente Teorema afferma che è sufficiente individuare un algoritmo risolutivo di complessità polinomiale per un solo problema NPC per poter dimostrare automaticamente che ogni problema NP è risolubile in tempo polinomiale e che pertanto P $=$ NP. Al tempo stesso, se per un qualsiasi problema NPC si riuscisse a dimostrare l'impossibilità di costruire un algoritmo risolutivo di complessità polinomiale, si potrebbe concludere che P $\neq$ NP e che P $\subset$ NP. Al momento la questione è ancora aperta e in attesa di una risposta.

Nel seguito saranno proposti diversi problemi di ottimizzazione combinatoria che, nella loro formulazione più generale, rientrano nella vasta classe dei problemi NP-completi. Concludiamo definendo un'ultima classe di complessità: la classe NPH dei **problemi NP-hard**. Questo insieme è costituito da tutti quei problemi che soddisfano la seconda proprietà della definizione della classe NPC, ma non la prima. Sono i problemi tra i più difficili, a cui si può ridurre in tempo polinomiale qualunque

problema NP, ma di cui non si conosce neanche un algoritmo di verifica di complessità polinomiale. In Figura 4 è riportato un diagramma che rappresenta le relazioni esistenti tra le classi di complessità definite nelle pagine precedenti.

Riferimenti bibliografici

[1] Giorgio Ausiello, Fabrizio D'Amore, Giorgio Gambosi, *Linguaggi, modelli, complessità*, Franco Angeli, Milano, 2003.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduzione agli algoritmi e strutture dati*, terza edizione, McGraw-Hill, 2009.

[3] Pierluigi Crescenzi, Giorgio Gambosi, Roberto Grossi, *Strutture di dati e algoritmi*, Pearson–Addison Wesley, 2006.

[4] Nigel J. Cutland, *Computability – An introduction to recessive function theory*, Cambridge University Press, 1980.

[5] Edsger W. Dijkstra, *A discipline of programming*, Prentice Hall, 1976.

[6] Marcello Frixione, Dario Palladino, *Funzioni, macchine, algoritmi – Introduzione alla teoria della computabilità*, Carocci, Roma, 2004.

[7] Michael R. Garey, David S. Johnson, *Computers and Intractability – A Guide to the Theory of NP-Completeness*, Freeman, 2003 (24^{th} Printing).

[8] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, *Automi, linguaggi e calcolabilità*, Addison-Wesley, Milano, 2003.

[9] Donald E. Knuth, *Stable Marriage and Its Relation to Other Combinatorial Problems*, American Mathematical Society, Montreal, 1997.

[10] Marco Liverani, *Programmare in C*, Seconda edizione, Esculapio, Bologna, 2013.

[11] Marco Liverani, *Qual è il problema? Metodi, strategie risolutive, algoritmi*, Mimesis, Milano, 2005.

[12] Fabrizio Luccio, *La struttura degli algoritmi*, Boringhieri, Torino, 1982.