Computabilità e complessità

I problemi computazionali possono essere classificati in base alla complessità dei relativi algoritmi di risoluzione. Questo capitolo offre una visione d'insieme dei temi che riguardano lo studio degli algoritmi e la difficoltà computazionale intrinseca dei problemi computazionali.

Indirizziamo al lettore alcune sfide su **problemi computazionali**, ovvero problemi che vorremmo risolvere al calcolatore mediante **algoritmi**:[1] l'algoritmo è l'essenza computazionale di un programma ma non deve essere identificato con quest'ultimo, in quanto un programma si limita a codificare in uno specifico linguaggio (di programmazione) i passi descritti da un algoritmo, e programmi diversi possono realizzare lo stesso algoritmo. Avendo cura di distillare gli aspetti rilevanti ai fini computazionali (trascendendo quindi dal particolare linguaggio, ambiente di sviluppo o sistema operativo adottato), possiamo discutere di algoritmi senza addentrarci nel gergo degli *hacker* e dei *geek*, rendendo così alla portata di molti, concetti utili a programmare in situazioni reali complesse.

La progettazione di algoritmi a partire da problemi provenienti dal mondo reale (il cosiddetto *problem solving*) è un processo *creativo* e *gratificante*. Gli ingredienti sono semplici, ovvero **array**, **liste**, **alberi** e **grafi** come illustrato nella Figura 1, ma essi ci permettono di strutturare i dati elementari (caratteri, interi, reali e stringhe) in forme più complesse, in modo da rappresentare le istanze di problemi computazionali *reali e concreti* nel mondo *virtuale* del calcolatore. Pur sembrando sorprendente che problemi computazionali complessi, come il calcolo delle previsioni metereologiche o la programmazione di un satellite, possano essere ricondotti all'elaborazione di soli quattro ingredienti di base, ricordiamo che l'informatica, al pari delle altre scienze quali la chimica, la fisica e la matematica, cerca di ricondurre i fenomeni (computazionali) a pochi elementi fondamentali.

Nel caso dell'informatica, come molti sanno, l'elemento costituente dell'informazione è il **bit**, componente reale del nostro mondo fisico quanto l'atomo o il quark: il bit rappresenta l'informazione minima che può assumere due soli valori (per esempio, `0`/`1`, acceso/spento, destra/sinistra o testa/croce). Negli anni '50, lavorando presso i prestigiosi laboratori di ricerca della compagnia telefonica statunitense AT&T Bell Labs,[2] il padre della teoria dell'informazione, Claude Shannon, definì il bit come la quantità di informazione necessaria a rappresentare un evento con due possibilità equiprobabili, e introdusse l'**entropia** come una misura della quantità minima di bit necessaria a rappresentare un contenuto informativo senza perdita di informazione (se possiamo memorizzare un film in un supporto digitale, o se possiamo ridurre il costo per bit di certi servizi di telefonia cellulare, lo dobbiamo a questo tipo di studi che hanno permesso l'avanzamento della tecnologia).

---

[1] Il termine "algoritmo" deriva dalla traslitterazione latina *Algorismus* del nome del matematico persiano del IX secolo, Muhammad al-Khwarizmi, che descrisse delle procedure per i calcoli aritmetici. Da notare che un algoritmo non necessariamente richiede la sua esecuzione in un calcolatore, ma può essere implementato, per esempio, mediante un dispositivo meccanico, un circuito elettronico o un sistema biologico.

[2] Presso gli stessi laboratori furono sviluppati, tra gli altri, il sistema operativo UNIX e i linguaggi di programmazione C e C++, in tempi successivi.

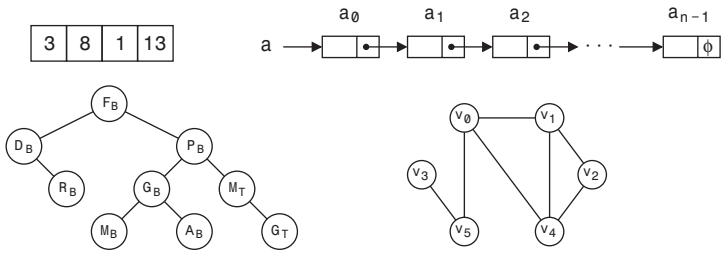


Figura 1 Ingredienti di base di un algoritmo: array, liste, alberi e grafi.

L'informazione può essere quindi misurata come le altre entità fisiche e, come quest'ultime, è sempre esistita: la fondamentale scoperta nel 1953 della "doppia elica" del DNA (l'acido desossiribonucleico presente nel nucleo di tutte le cellule), da parte di James Watson e Francis Crick, ha infatti posto le basi biologiche per la comprensione della struttura degli esseri viventi da un punto di vista "informativo". Il DNA rappresenta in effetti l'informazione necessaria alle funzionalità degli esseri viventi e può essere rappresentato all'interno del calcolatore con le strutture di dati elencate sopra. In particolare, la doppia elica del DNA è costituita da due filamenti accoppiati e avvolti su se stessi, a formare una struttura elicoidale tridimensionale. Ciascun filamento può essere ricondotto a una *sequenza* (e, quindi, a un *array* oppure a una *lista*) di acidi nucleici (adenina, citosina, guanina e timina) chiamata struttura primaria: per rappresentare tale sequenza, usiamo un alfabeto finito come nei calcolatori, quaternario invece che binario, dove le lettere sono scelte tra le iniziali delle quattro componenti fondamentali: {A, C, G, T}. La sequenza di acidi nucleici si ripiega su se stessa a formare una struttura secondaria che possiamo rappresentare come un *albero*. Infine, la struttura secondaria si dispone nello spazio in modo da formare una struttura terziaria elicoidale, che possiamo modellare come un *grafo*.

1 Indecidibilità di problemi computazionali

Nel libro *Algorithmics: The Spirit of Computing*, l'autore David Harel riporta un estratto di un articolo della rivista *Time Magazine* di diversi anni fa in cui il redattore di un periodico specializzato in informatica dichiarava che il calcolatore può fare qualunque cosa: basta scrivere il programma adatto a tale scopo, in quanto le eventuali limitazioni sono dovute all'architettura del calcolatore (per esempio, la memoria disponibile), e non certo al programma eseguito. Probabilmente al redattore sfuggiva l'esistenza del **problema della fermata**, pubblicato nel 1937 dal matematico inglese Alan Turing, uno dei padri dell'informatica che, con la

sua idea di macchina universale, è stato il precursore del moderno concetto di "software".[3]

Espresso in termini odierni, il problema della fermata consiste nel capire se un generico programma **termina** (ovvero, finisce la sua esecuzione) oppure "va in ciclo" (ovvero, continua a ripetere sempre la stessa sequenza di istruzioni all'infinito), supponendo di non avere limiti di tempo e di memoria per il calcolatore impiegato a tal proposito. Per esempio, consideriamo il problema di stabilire se un dato intero $p > 1$ è un numero primo, ovvero è divisibile soltanto per 1 e per se stesso: 2, 3, 5, 7, 11, 13 e 17 sono alcuni numeri primi (tra l'altro, trovare grandi numeri primi è alla base di diversi protocolli crittografici). Il seguente programma codifica un possibile algoritmo di risoluzione per tale problema.

```
1  Primo( numero ):                                  ⟨pre: numero > 1⟩
2    fattore = 2;
3    WHILE (numero % fattore != 0 )
4      fattore = fattore + 1;
5    RETURN (fattore == numero);
```

Tale codice non è particolarmente efficiente: per esempio, potremmo evitare di verificare che `numero` sia divisibile per `fattore` quando quest'ultimo è pari. Tuttavia, siamo sicuri che esso termina perché la variabile `fattore` viene incrementata a ogni iterazione e la guardia del ciclo nella riga 3 viene sicuramente verificata se `fattore` è uguale a `numero`.

Nel semplice caso appena discusso, decidere se il programma termina è quindi immediato. Purtroppo, non è sempre così, come mostrato dal seguente programma il cui scopo è quello di trovare il più piccolo numero intero pari che *non* sia la somma di due numeri primi.

```
1  CongetturaGoldbach( ):
2    n = 2;
3    DO {
4      n = n + 2;
5      controesempio = TRUE;
6      FOR (p = 2; p <= n-2; p = p + 1) {
7        q = n - p;
8        IF (Primo(p) && Primo(q)) controesempio = FALSE
9      }
10   } WHILE (!controesempio);
11   RETURN n;
```

[3] Per quanto riguarda il problema della fermata, lo stesso Turing nel suo lavoro del 1937 afferma di essersi ispirato al primo teorema di incompletezza di Kurt Gödel, il quale asserisce che esistono teoremi veri ma indimostrabili in qualunque sistema formale che possa descrivere l'aritmetica degli interi.

Se fossimo in grado di decidere se la funzione `CongetturaGoldbach` termina o meno, allora avremmo risolto la *congettura di Goldbach*, formulata nel XVIII secolo, la quale afferma che ogni numero intero `n` ≥ 4 pari è la somma di due numeri primi `p` e `q`. In effetti, il programma tenta di trovare un valore di `n` per cui la congettura non sia vera: se la congettura di Goldbach è però vera, allora il programma non termina mai (ipotizzando di avere tutto lo spazio di memoria necessario). Nonostante il premio milionario offerto nel 2000 dalla casa editrice britannica Faber&Faber a chi risolvesse la congettura, nessuno è stato in grado ancora di provarla o di trovarne un controesempio.

Riuscire a capire se un programma arbitrario termina non è soltanto un'impresa ardua (come nel caso del precedente programma) ma, in generale, è *impossibile* per i calcolatori, come Turing ha dimostrato facendo riferimento al problema della fermata e usando le **macchine di Turing** (un formalismo alternativo a quello adottato in questo libro).

Ricordiamo che, nei calcolatori, un programma è codificato mediante una sequenza di simboli che viene data in ingresso a un altro programma (tipicamente un compilatore): non deve quindi stupirci il fatto che una stessa sequenza di simboli possa essere interpretata sia come un programma che come un dato d'ingresso di un altro programma.

Quest'osservazione è alla base del risultato di Turing, la cui dimostrazione procede per assurdo. Supponiamo che esista un programma `Termina(A,D)`, il quale, preso un programma `A` e i suoi dati in ingresso `D`, restituisce (in tempo finito) un valore di verità per indicare che `A` termina o meno quando viene eseguito sui dati d'ingresso `D`.

Notiamo che sia `A` che `D` sono sequenze di simboli, e siamo noi a stabilire che `A` debba essere intesa come un programma mentre `D` come i suoi dati d'ingresso: è quindi perfettamente legittimo invocare `Termina}(A,A)`, come accade all'interno del seguente programma.

```
1  Paradosso( A ):
2    WHILE (Termina( A, A ))
3    ;
```

Poiché il corpo del ciclo `WHILE` è vuoto, per ogni programma `A`, osserviamo che `Paradosso(A)` termina se e solo se la guardia `Termina(A,A)` restituisce il valore `FALSE`, ovvero se e solo se il programma `A` non termina quando viene eseguito sui dati d'ingresso `A`. Possiamo quindi concludere che `Paradosso(Paradosso)` termina se e solo se la guardia `Termina(Paradosso,Paradosso)` restituisce il valore `FALSE`, ovvero se e solo se il programma `Paradosso` non termina quando viene eseguito sui dati d'ingresso `Paradosso`. In breve, `Paradosso(Paradosso)` termina se e solo se `Paradosso(Paradosso)` non termina!

Questa contraddizione deriva dall'aver assunto l'esistenza di `Termina`, l'unico anello debole del filo logico tessuto nell'argomentazione precedente. Quindi, un tale programma non può esistere e, pertanto, diciamo che il problema della fermata è **indecidibile**. Purtroppo esso non è l'unico: per esempio, stabilire se due programmi `A` e `B` sono equivalenti, ovvero producono sempre i medesimi risultati a parità di dati in ingresso, è anch'esso un problema indecidibile. Notiamo che l'uso di uno specifico linguaggio non influisce su tali risultati di indecidibilità, i quali valgono per qualunque modello di calcolo che possa formalizzare il comportamento di un calcolatore (più precisamente di una macchina di Turing).

2 Trattabilità di problemi computazionali

L'esistenza di problemi indecidibili restringe la possibilità di progettare algoritmi e programmi ai soli problemi decidibili. In questo ambito, non tutti i problemi risultano risolvibili in tempo ragionevole, come testimoniato dal noto problema delle **Torri di Hanoi**, un gioco del XIX secolo inventato da un matematico francese, Edouard Lucas, legandolo alla seguente leggenda indiana (probabilmente falsa) sulla divinità Brahma e sulla fine del mondo. In un tempio induista dedicato alla divinità, vi sono tre pioli di cui il primo contiene $n = 64$ dischi d'oro impilati in ordine di diametro decrescente, con il disco più ampio in basso e quello più stretto in alto (gli altri due pioli sono vuoti). Dei monaci *sannyasin* spostano i dischi dal primo al terzo piolo usando il secondo come appoggio, con la regola di non poter spostare più di un disco alla volta e con quella di non porre mai un disco di diametro maggiore sopra un disco di diametro inferiore. Quando i monaci avranno terminato di spostare tutti i dischi nel terzo piolo, avverrà la fine del mondo.

La soluzione di questo gioco è semplice da descrivere usando la ricorsione. Supponiamo di avere spostato ricorsivamente i primi $n - 1$ dischi sul secondo piolo, usando il terzo come appoggio. Possiamo ora spostare il disco più grande dal primo al terzo piolo, e quindi ricorsivamente spostare gli $n - 1$ dischi dal secondo al terzo piolo usando il primo come appoggio.

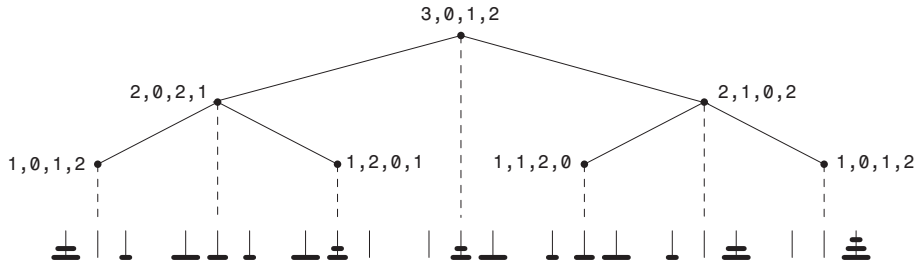
```
1  TorriHanoi( n, s, a, d ):
2    IF (n = 1) {
3      PRINT s ↦ d;
4    } ELSE {
5      TorriHanoi( n - 1, s, d, a );
6      PRINT s ↦ d;
7      TorriHanoi( n - 1, a, s, d );
8    }
```

Dimostriamo, per induzione sul numero n di dischi, che il numero di mosse effettuate eseguendo tale programma e stampate come "origine $\mapsto$ destinazione", è pari a $2^n - 1$: il caso base $n = 1$ è immediato; nel caso $n > 1$ occorrono $2^{n-1} - 1$

mosse per ciascuna delle due chiamate ricorsive per ipotesi induttiva, a cui aggiungiamo la mossa nella riga 6, per un totale di $2 \times (2^{n-1} - 1) + 1 = 2^n - 1$ mosse.

ESEMPIO 1

Consideriamo il caso in cui n = 3. Nella figura seguente, mostriamo le chiamate ricorsive effettuate dall'esecuzione dell'algoritmo, indicando per ogni chiamata i valori dei suoi quattro argomenti, e la sequenza delle sette mosse eseguite per spostare i tre dischi dal paletto a sinistra al paletto a destra.



Purtroppo, non c'è speranza di trovare un programma che effettui un numero inferiore di mosse, in quanto è stato dimostrato che le $2^n - 1$ mosse sono necessarie e non è possibile impiegarne di meno. Nel problema originale con n = 64 dischi, supponendo che ogni mossa richieda un secondo, occorrono $2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$ secondi, che equivalgono a circa 584 942 417 355 anni, ovvero quasi 585 miliardi di anni: per confronto, la teoria del *big bang* asserisce che l'Universo è stato creato da un'esplosione cosmica in un periodo che risale a circa 10-20 miliardi di anni fa.

Le Torri di Hanoi mostrano dunque che, anche se un problema è decidibile ovvero è risolubile mediante un algoritmo, non è detto che l'algoritmo stesso possa sempre risolverlo in tempi ragionevoli: ciò è dovuto al fatto che il numero di mosse e quindi il tempo di esecuzione del programma, è **esponenziale** nel numero n di dischi (n appare all'esponente di $2^n - 1$). Il tempo necessario per spostare i dischi diventa dunque rapidamente insostenibile, anche per un numero limitato di dischi, come illustrato nella seguente tabella, in cui il tempo di esecuzione è espresso in secondi (s), minuti (m), ore (h), giorni (g) e anni (a).

n	5	10	15	20	25	30	35	40	45
tempo	31 s	17 m	9 h	12 g	1 a	34 a	1089 a	34865 a	1115689 a

L'esponenzialità del tempo di esecuzione rende anche limitato l'effetto di eventuali miglioramenti nella velocità di esecuzione dei singoli passi, perché, in tal caso, basta aumentare di poco il numero n di dischi per vanificare ogni miglioramento. Supponiamo infatti di poter eseguire $m = 2^s$ operazioni in un secondo, invece di una singola operazione al secondo: in tal caso, anziché circa 2^n secondi,

ne occorrono $2^n/m = 2^{n-s}$ per spostare gli n dischi. L'effetto di tale miglioramento viene però neutralizzato molto rapidamente al crescere del numero di dischi in quanto è sufficiente portare tale numero a n + s (dove s = log m) per ottenere lo stesso tempo complessivo di esecuzione. In altre parole, un miglioramento delle prestazioni per un fattore *moltiplicativo* si traduce in un aumento solo *additivo* del numero di dischi trattabili. La tabella seguente esemplifica questo comportamento nel caso n = 64, mostrando il numero di dischi gestibili in un tempo pari a 18 446 744 073 709 551 615 secondi, al variare della velocità di esecuzione: come possiamo vedere, miglioramenti anche molto importanti di quest'ultima si traducono in piccoli incrementi del numero di dischi che il programma è in grado di gestire.

operazioni/sec	1	10	100	10^3	10^4	10^5	10^6	10^9
numero dischi	64	67	70	73	77	80	83	93

Di diversa natura è invece l'andamento **polinomiale**, come possiamo mostrare se consideriamo la generalizzazione del problema delle Torri di Hanoi al caso in cui siano disponibili, oltre al piolo sorgente e a quello destinazione, k > 1 pioli di appoggio. A tale scopo, supponiamo che i pioli di appoggio siano numerati da 1 a k e che il problema consista nello spostare i dischi dal piolo 0 al piolo k + 1 (rispettando le regole sopra descritte). Se k = n – 1, allora il problema può essere facilmente risolto posizionando i primi n – 1 dischi negli n – 1 pioli di appoggio, spostando il disco più grande nel piolo destinazione e, infine, spostando i rimanenti n – 1 dai pioli di appoggio a quello destinazione, rispettando la loro grandezza. Tutto ciò è formalizzabile nel codice seguente.

```
1  TorriHanoiLineare( n ):                          ⟨pre: n > 0 e n + 1 pioli⟩
2    FOR (i = 1; i <= n-1; i = i+1)
3      PRINT 0 ↦ i;
4    PRINT 0 ↦ n;
5    FOR (i = n-1; i >= 1; i = i-1)
6      PRINT i ↦ n;
```

In tal caso, il numero di mosse eseguite è pari a (n – 1) + 1 + (n –1) = 2n – 1, ovvero è **lineare** nel numero di dischi (ovviamente al prezzo di un pari numero di pioli). Il passaggio da un andamento esponenziale a uno lineare ha due importanti conseguenze. In primo luogo, una funzione lineare cresce molto più lentamente di una qualunque funzione esponenziale, come mostrato nella seguente tabella relativa alla funzione 2n – 1 e analoga a quella vista nel caso della funzione 2^n.

n	5	10	15	20	25	30	35	40	45
tempo	9 s	19 s	29 s	39 s	49 s	59 s	69 s	79 s	89 s

In secondo luogo, la linearità del tempo di esecuzione rende molto più effica-

ci gli eventuali miglioramenti nella velocità di esecuzione dei singoli passi. Per esempio, nel caso del problema generalizzato delle Torri di Hanoi con n dischi e n – 1 pioli di appoggio, potendo eseguire m operazioni in un secondo, invece di una singola operazione al secondo, occorrerebbero $\frac{2n-1}{m} \leq 2n/m$ secondi per spostare gli n dischi. L'effetto di tale miglioramento permane a lungo in quanto è necessario portare il numero di dischi a circa $n \times m$ per ottenere lo stesso tempo complessivo di esecuzione. In altre parole, un miglioramento di un fattore *moltiplicativo* nelle prestazioni si traduce in un aumento anch'esso *moltiplicativo* del numero di dischi trattabili.

Per concludere questa breve trattazione del problema delle Torri di Hanoi, mostriamo come sia possibile utilizzare un numero minore di pioli di appoggio pur mantenendo il numero di mosse eseguite polinomiale (in particolare quadratico) rispetto al numero di dischi. L'idea dell'algoritmo è la seguente. Avendo a disposizione k pioli di appoggio, possiamo dividere gli n dischi in k gruppi di n/k dischi ciascuno (per semplicità, assumiamo che k sia un divisore di n), posizionare questi gruppi nei k pioli, utilizzando l'algoritmo TorriHanoi e il piolo destinazione come appoggio, e spostarli poi (in ordine inverso) nel piolo destinazione, utilizzando nuovamente l'algoritmo TorriHanoi e il piolo sorgente come appoggio. Tale algoritmo è mostrato nel codice seguente (non è difficile estendere tale codice in modo che funzioni per tutti i valori di n, anche quando n non è un multiplo di k).

```
1  TorriHanoiGen( n, k ):                    ⟨pre: n > 0 multiplo di k⟩
2    FOR (i = 1; i <= k; i = i+1)
3      TorriHanoi(n/k, 0, k+1, i);
4    FOR (i = k; i >= 1; i = i-1)
5      TorriHanoi(n/k, i, 0, k+1);
```

In questo caso, il numero totale di mosse è dunque pari a 2k volte il numero di mosse richiesto per spostare n/k dischi usando tre pioli, ovvero è pari a $2k(2^{n/k} - 1)$. Assumendo $n \geq 2$ e ponendo $k = \left\lceil \frac{n}{\log n} \right\rceil$ (per cui $\frac{n}{\log n} \leq k \leq \frac{n}{\log n} + 1$), abbiamo che il numero totale di mosse è al più

$$2\left(\frac{n}{\log n} + 1\right)\left(2^{\frac{n}{n/\log n}} - 1\right) = 2\left(\frac{n}{\log n} + 1\right)(n-1) < 2n^2 + 2n \leq 4n^2.$$

In tal caso, il problema delle Torri di Hanoi può quindi essere risolto con un numero di mosse quadratico nel numero dei dischi (notiamo che è in generale un problema aperto stabilire il numero minimo di mosse per ogni n e per ogni $k > 1$). Per esempio, volendo $4 \cdot 64^2 = 4 \cdot 4096 = 16384$ spostare gli $n = 64$ dischi del problema originale usando $k = 11$ pioli di appoggio, occorrono al più secondi contro i $2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$ secondi necessari nel caso di tre pioli (suppo-

nendo di poter effettuare una mossa al secondo). Ancora una volta, il passaggio da un andamento esponenziale a uno quadratico si riflette significativamente sia sul tempo di esecuzione che sull'efficacia di eventuali miglioramenti nella velocità di esecuzione dei singoli passi.

3 Rappresentazione e dimensione dei dati

Volendo generalizzare la discussione fatta nel caso delle Torri di Hanoi a un qualunque problema computazionale, è anzitutto necessaria una breve escursione sulla rappresentazione e sulla codifica dei dati elementari utilizzati dal calcolatore. Secondo quanto detto in riferimento alla teoria dell'informazione di Claude Shannon, il bit (*binary digit*) segnala la presenza (1) oppure l'assenza (0) di un segnale o di un evento con due possibilità equiprobabili.[4]

La stessa sequenza di bit può essere interpretata in molti modi, a seconda del significato che le vogliamo assegnare nel contesto in cui la usiamo: può essere del codice da eseguire oppure dei dati da elaborare, come abbiamo visto nel problema della fermata. In particolare, gli interi nell'insieme $\{0, 1, \ldots, 2^k - 1\}$ possono essere codificati con k bit $b_{k-1}b_{k-2}\cdots b_1b_0$. La regola per trasformare tali bit in un numero intero è semplice: basta moltiplicare ciascuno dei bit per potenze crescenti di 2, a partire dal bit meno significativo b_0, ottenendo $\sum_{i=0}^{k-1} b_i \times 2^i$. Per esempio, la sequenza 0101 codifica il numero intero $5 = 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$. La regola inversa può essere data in vari modi, e l'idea è quella di sottrarre ogni volta la massima potenza del 2 fino a ottenere 0. Per rappresentare sia numeri positivi che negativi è sufficiente aggiungere un bit di segno.

I caratteri sono codificati come interi su $k = 8$ bit (ASCII) oppure su $k = 16$ bit (*Unicode*/UTF8). La codifica riflette l'ordine alfabetico, per cui la lettera 'A' viene codificata con un intero più piccolo della lettera 'Z' (bisogna porre attenzione al fatto che il carattere '7' non è la stessa cosa del numero 7). Le stringhe sono sequenze di caratteri alfanumerici che vengono perciò rappresentate come sequenze di numeri terminate da un carattere speciale oppure a cui vengono associate le rispettive lunghezze.

I numeri reali sono codificati con un numero limitato di bit a precisione finita di 32 o 64 bit nello standard IEEE754 (quindi sono piuttosto dei numeri razionali). Il primo bit è utilizzato per il segno; un certo numero di bit successivi codifica l'esponente, mentre il resto dei bit serve per la mantissa. Per esempio, la codifica di $-0{,}275 \times 2^{18}$ è ottenuta codificando il segno meno, l'esponente 18, e quindi la mantissa 0,275 (ciascuno con il numero assegnato di bit).
Infine, in generale, un insieme finito è codificato come una sequenza di elementi separati da un carattere speciale per quell'insieme: questa codifica ci permetterà,

[4] Il bit viene usato come unità di misura: 1 *byte* = 8 bit, 1 *kilobyte* (KB) = 210 byte = 1024 byte, 1 *megabyte* (MB) = 210 KB = 1 048 576 byte, 1 *gigabyte* (GB) = 210 MB = 1 073 741 824 byte, 1 *terabyte* (TB) = 210 GB, 1 *petabyte* (PB) = 210 TB e così via.

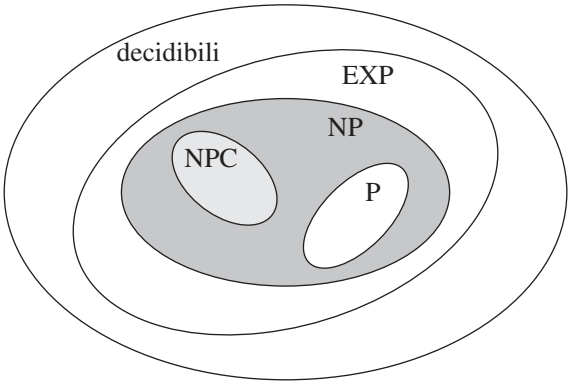


Figura 2 Una prima classificazione dei problemi computazionali decidibili.

se necessario, di codificare anche insiemi di insiemi, usando gli opportuni caratteri speciali di separazione.

Le regole di codifica discusse finora, ci consentono, per ogni dato, di ricavarne una rappresentazione binaria: nel definire la **dimensione del dato**, faremo riferimento alla lunghezza di tale rappresentazione o a una misura equivalente.

4 Algoritmi polinomiali ed esponenziali

Abbiamo già osservato che, tranne che per piccole quantità di dati, un algoritmo che impiega un numero di passi esponenziale è impossibile da usare quanto un algoritmo che non termina! Nel seguito useremo il termine **algoritmo polinomiale** per indicare un algoritmo, per il quale esiste una costante `c > 0`, il cui numero di passi elementari sia al massimo pari a `nc` per ogni dato in ingresso di dimensione `n`. Questa definizione ci porta a una prima classificazione dei problemi computazionali come riportato nella Figura 2 dove, oltre alla divisione in problemi indecidibili e decidibili, abbiamo l'ulteriore suddivisione di questi ultimi in **problemi trattabili** (per i quali esiste un algoritmo risolutivo polinomiale) e **problemi intrattabili** (per i quali un tale algoritmo non esiste): facendo riferimento alla figura, tali classi di problemi corrispondono rispettivamente a P e EXP – P, dove EXP rappresenta la classe di problemi risolubili mediante un **algoritmo esponenziale**, ovvero un algoritmo il cui numero di passi è al più esponenziale nella dimensione del dato in ingresso.[5] Talvolta gli algoritmi esponenziali sono utili per esaminare le caratteristiche di alcuni problemi combinatori sulla base della generazione esaustiva di tutte le istanze di piccola taglia.

[5] Volendo essere più precisi, i problemi intrattabili sono tutti i problemi decidibili che non sono inclusi in PP: tra di essi, quindi, vi sono anche problemi che non sono contenuti in EXP. Nel resto di questo libro, tuttavia, non considereremo mai problemi che non ammettano un algoritmo esponenziale.

Discutiamo un paio di casi, che rappresentano anche un ottimo esempio di uso della ricorsione nella risoluzione dei problemi computazionali. Nel primo esempio, vogliamo generare tutte le 2^n sequenze binarie di lunghezza n, che possiamo equivalentemente interpretare come tutti i possibili sottoinsiemi di un insieme di n elementi. Per illustrare questa corrispondenza, numeriamo gli elementi da 0 a n – 1 e associamo il bit in posizione b della sequenza binaria all'elemento b dell'insieme fornito (dove 0 ≤ b ≤ n – 1): se tale bit è pari a 1, l'elemento b è nel sottoinsieme così rappresentato; altrimenti, il bit è pari a 0 e l'elemento non appartiene a tale sottoinsieme. Durante la generazione delle 2^n sequenze binarie, memorizziamo ciascuna sequenza binaria A e utilizziamo la procedura Elabora per stampare A o per elaborare il corrispondente sottoinsieme. Notiamo che A viene riutilizzata ogni volta sovrascrivendone il contenuto ricorsivamente: il bit in posizione b, indicato con A[b – 1], deve valere prima 0 e, dopo aver generato tutte le sequenze con tale bit, deve valere 1, ripetendo la generazione. Il seguente codice ricorsivo permette di ottenere tutte le 2^n sequenze binarie di lunghezza n: inizialmente, dobbiamo invocare la funzione GeneraBinarie con input b = n.

```
1  GeneraBinarie( A, b ):          ⟨pre: i primi b bit in A sono da generare⟩
2    IF (b == 0) {
3      Elabora( A );
4    } ELSE {
5      A[b-1] = 0;
6      GeneraBinarie( A, b-1 );
7      A[b-1] = 1;
8      GeneraBinarie( A, b-1 );
9    }
```

ESEMPIO 2

Considerando il caso n = 4, l'algoritmo di generazione delle sequenze binarie produce prima tutte le sequenze che finiscono con 0 per poi passare a produrre tutte quelle che finiscono con 1. Lo stesso principio viene applicato ricorsivamente alle sequenze di lunghezza 3, 2 e 1 ottenendo quindi tutte le sequenze binarie di quattro simboli nel seguente ordine:

```
0000    1000    0100    1100
0010    1010    0110    1110
0001    1001    0101    1101
0011    1011    0111    1111
```

Il secondo esempio di un utile algoritmo esponenziale riguarda la generazione delle permutazioni degli n elementi contenuti in una sequenza A. Ciascuno degli n

elementi occupa, a turno, l'ultima posizione in A e i rimanenti n – 1 elementi sono ricorsivamente permutati. Per esempio, volendo generare tutte le permutazioni di n = 4 elementi a, b, c, d in modo sistematico, possiamo generare prima quelle aventi d in ultima posizione (elencate nella prima colonna), poi quelle aventi c in ultima posizione (elencate nella seconda colonna) e così via:

```
a b c d     a b d c     a d c b     d b c a
b a c d     b a d c     d a c b     b d c a
a c b d     a d b c     a c d b     d c b a
c a b d     d a b c     c a d b     c d b a
c b a d     d b a c     c d a b     c b d a
b c a d     b d a c     d c a b     b c d a
```

Restringendoci alle permutazioni aventi d in ultima posizione (prima colonna), possiamo permutare i rimanenti elementi a, b, c in modo analogo usando la ricorsione su questi tre elementi. A tal fine, notiamo che le permutazioni generate per i primi n – 1 = 3 elementi, sono identiche a quelle delle altre tre colonne mostrate sopra. Per esempio, se ridenominiamo l'elemento c (nella prima colonna) con d (nella seconda colonna), otteniamo le *medesime* permutazioni di n – 1 = 3 elementi; analogamente, possiamo ridenominare gli elementi b e d (nella seconda colonna) con d e c (nella terza colonna), rispettivamente. In generale, le permutazioni di n – 1 elementi nelle colonne sopra possono essere messe in corrispondenza biunivoca e, pertanto, ciò che conta è il numero di elementi da permutare come riportato nel codice seguente. Invocando tale codice con parametro d'ingresso p = n, possiamo ottenere tutte le n! permutazioni degli elementi in A:

```
1  GeneraPermutazioni( A, p ):   ⟨pre: i primi p elementi di A sono da permutare⟩
2    IF (p == 0) {
3      Elabora( A );
4    } ELSE {
5      FOR (i = p-1; i >= 0; i = i-1) {
6        Scambia( i, p-1 );
7        GeneraPermutazioni( A, p-1 );
8        Scambia( i, p-1 );
9      }
10   }
```

Notiamo l'utilizzo di una procedura Scambia prima e dopo la ricorsione così da mantenere l'invariante che gli elementi, dopo esser stati permutati, vengono riportati al loro ordine di partenza, come può essere verificato simulando l'algoritmo suddetto.

5 Problemi NP-completi

La classificazione dei problemi decidibili nella Figura 2 ha in realtà una zona grigia localizzata tra i problemi trattabili e quelli intrattabili (le definizioni rigorose saranno date nell'ultimo capitolo del libro). Esistono decine di migliaia di esempi interessanti di problemi che giacciono in tale zona grigia: di questi ne riportiamo uno tratto dal campo dei solitari e relativo al noto gioco del *Sudoku*.

In tale solitario, il giocatore è posto di fronte a una tabella di nove righe e nove colonne parzialmente riempita con numeri compresi tra 1 e 9, come nell'istanza mostrata nella parte sinistra della Figura 3. Come possiamo vedere, la tabella è suddivisa in nove sotto-tabelle, ciascuna di tre righe e tre colonne. Il compito del giocatore è quello di riempire le caselle vuote della tabella con numeri compresi tra 1 e 9, rispettando i seguenti vincoli:

1. ogni riga contiene tutti i numeri compresi tra 1 e 9;
2. ogni colonna contiene tutti i numeri compresi tra 1 e 9;
3. ogni sotto-tabella contiene tutti i numeri compresi tra 1 e 9.

Nella parte destra della Figura 3 mostriamo una soluzione ottenuta abbastanza facilmente sulla base di implicazioni logiche del tipo: "visto che la sotto-tabella in alto a destra deve contenere un 3, che la prima riga e la seconda riga contengono un 3 e che la nona colonna contiene un 3, allora nella casella in terza riga e settima colonna ci deve essere un 3". Tali implicazioni consentono al giocatore di determinare *inequivocabilmente* il contenuto di una casella: notiamo che, nel caso mostrato nella figura, in ogni passo del processo risolutivo, vi è sempre almeno una casella il cui contenuto può essere determinato sulla base di siffatte implicazioni.

Tuttavia, le configurazioni iniziali che vengono proposte al giocatore non sono sempre di tale livello di difficoltà: le configurazioni più difficili raramente consentono di procedere in modo univoco fino a raggiungere la soluzione finale, costringendo pertanto il giocatore a operare delle scelte che possono talvolta

3	9							8
	7	1			3			
		8		4	9		6	
1			2	7				9
6								3
5				3	6			4
	4		1	5		9		
			9			8	2	
9							4	7

3	9	6	5	1	2	4	7	8
4	7	1	6	8	3	5	9	2
2	5	8	7	4	9	3	6	1
1	3	4	2	7	5	6	8	9
6	8	7	4	9	1	2	5	3
5	2	9	8	3	6	7	1	4
8	4	2	1	5	7	9	3	6
7	1	3	9	6	4	8	2	5
9	6	5	3	2	8	1	4	7

Figura 3 Un esempio di istanza del gioco del Sudoku e la corrispondente soluzione.

			6		2		9	
								6
			7	3	1	5		8
4		9	3			6		5
		3				1		
5		8			7	9		2
		1	5	2	3			
7								
	6	2	9		4			

			6		2		9	
			8					6
			7	3	1	5		8
4	2	9	3	1	8	6	7	5
6	7	3	2			1	8	4
5	1	8	4	6	7	9	3	2
		1	5	2	3			
7			1	8	6			
	6	2	9	7	4			

Figura 4 Un esempio di istanza difficile del Sudoku e una successiva configurazione senza proseguimento univoco.

rivelarsi sbagliate. Con riferimento alla Figura 4, possiamo procedere inizialmente in modo univoco partendo dalla configurazione nella parte sinistra fino a ottenere la configurazione nella parte destra: a questo punto, non esiste alcuna casella il cui contenuto possa essere determinato in modo univoco. Per esempio, la casella in basso a destra può contenere sia un 1 che un 3 e non abbiamo modo di scegliere quale valore includere, se non procedendo per tentativi e annullando le scelte parziali che conducano a un vicolo cieco.

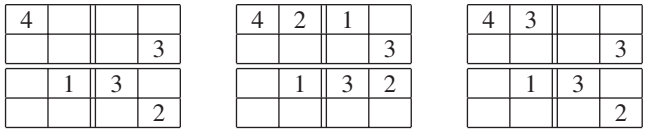
In questi casi, il giocatore è dunque costretto a eseguire un algoritmo di **backtrack**, in base al quale la scelta operata più recentemente (se non conduce a una soluzione del problema) viene annullata e sostituita con un'altra scelta possibile (che non sia già stata analizzata). Questo modo di procedere è formalizzato nella seguente funzione ricorsiva `Sudoku`, la quale esamina tutte le caselle inizialmente vuote, nell'ordine implicitamente specificato dalle funzioni `PrimaVuota`, `SuccVuota` e `UltimaVuota` (per esempio, scorrendo la tabella per righe o per colonne): supponendo che la configurazione iniziale contenga almeno una casella vuota, la funzione deve inizialmente essere invocata con argomento la casella restituita da `PrimaVuota`.

```
1  Sudoku( casella ):                                  ⟨pre: casella vuota⟩
2    elenco = insieme delle cifre ammissibili per casella;
3    FOR (i = 0; i < ¦elenco¦; i = i+1) {
4      Assegna( casella, elenco[i] );
5      IF (!UltimaVuota(casella) && !Sudoku(SuccVuota(casella))) {
6        Svuota( casella );
7      } ELSE {
8        RETURN TRUE;
9      }
10   }
11   RETURN FALSE;
```

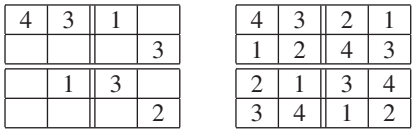
Per ogni casella vuota, il codice calcola l'elenco delle cifre (comprese tra 1 e 9) che in essa possono essere contenute (riga 2): prova dunque ad assegnare a tale casella una dopo l'altra tali cifre (riga 4). Se giunge all'ultima casella della tabella (riga 5), il codice restituisce il valore `TRUE` (riga 8):[6] in questo caso, una soluzione al problema è stata trovata. Altrimenti, invoca ricorsivamente la funzione `Sudoku` con argomento la prossima casella vuota e, nel caso in cui l'invocazione ricorsiva non abbia prodotto una soluzione accettabile, annulla la scelta appena fatta (riga 6) e ne prova un'altra. L'intero procedimento ha termine nel momento in cui il codice trova una soluzione oppure esaurisce le scelte possibili (riga 11).

ESEMPIO 3

Per mostrare un esempio di esecuzione di `Sudoku`, consideriamo una variante del gioco in cui la tabella abbia dimensione 4×4 e possa contenere solo numeri da 1 a 4. In particolare, prendiamo in esame la configurazione iniziale mostrata nella parte sinistra della seguente figura.



Il nostro algoritmo esamina per prima la cella in prima riga e seconda colonna: in tal caso, `elenco` include i valori 2 e 3. Per prima viene esplorata la possibilità di assegnare a tale cella il valore 2. Passando alla cella successiva, questa può contenere solo il valore 1, il quale viene quindi assegnato alla cella stessa ottenendo la configurazione mostrata al centro della figura. In tale configurazione, l'insieme `elenco` della cella in prima riga e quarta colonna è vuoto: l'algoritmo è quindi costretto a tornare indietro alla prima scelta effettuata ed esplorare l'unica altra scelta possibile, ovvero assegnare il valore 3 alla cella in prima riga e seconda colonna, ottenendo la configurazione mostrata nella parte destra della figura. Passando alla cella successiva, il corrispondente `elenco` contiene i valori 1 e 2. L'algoritmo assegna pertanto il valore 1 a tale cella, impedendo in tal modo di assegnare un valore alla cella successiva, come mostrato nella parte sinistra della figura seguente.



Quindi, ancora una volta l'algoritmo torna all'ultima scelta effettuata ed esplora la possibilità di assegnare il valore 2 alla cella in seconda riga e terza colonna. È facile a questo punto verificare che l'algoritmo può proseguire riempiendo l'intera tabella senza dover più ritornare sui propri passi modificando scelte già effettuate, e ottenendo la configurazione finale mostrata nella parte destra della figura.

[6] Notiamo che la congiunzione di due o più operandi booleani è valutata in modo pigro: gli operandi sono valutati da sinistra verso destra e la valutazione ha termine non appena viene incontrato un operando il cui valore sia `FALSE`. Inoltre, `RETURN` termina la chiamata di funzione restituendo il valore specificato.

Osserviamo che l'algoritmo sopra esposto esegue, nel caso pessimo, un numero di operazioni proporzionale a 9^m, $m \leq 9 \times 9$ indica il numero di caselle inizialmente vuote: infatti, per ogni casella vuota della tabella vi sono al più 9 possibili cifre con cui tentare di riempire tale casella. In generale, usando n cifre (con n numero quadrato arbitrariamente grande), il gioco necessita di una tabella di dimensione $n \times n$, e quindi l'algoritmo suddetto ha complessità esponenziale, in quanto richiede circa $n^m \leq n^{n \times n} = 2^{n^2 \log n}$ operazioni.

A differenza del problema delle Torri di Hanoi, non possiamo però concludere che il problema del gioco del Sudoku sia intrattabile: nonostante si conoscano solo algoritmi esponenziali per il Sudoku, nessuno finora è riuscito a dimostrare che tale problema possa ammettere o meno una risoluzione mediante algoritmi polinomiali. Un'evidenza della diversa natura dei due problemi dal punto di vista della complessità computazionale, deriva dal fatto che, quando il secondo problema ammette una soluzione, esiste sempre una prova dell'esistenza di una tale soluzione che possa essere verificata in tempo polinomiale (al contrario, non esiste alcun algoritmo polinomiale di verifica per il problema delle Torri di Hanoi).

Supponiamo infatti che, stanchi di tentare di riempire una tabella di dimensione $n \times n$ pubblicata su una rivista di enigmistica, incominciamo a nutrire dei seri dubbi sul fatto che tale tabella ammetta una soluzione. Per tale motivo, decidiamo di rivolgerci direttamente all'editore chiedendo di convincerci che è possibile riempire la tabella. Ebbene, l'editore ha un modo molto semplice di fare ciò, inviandoci la sequenza delle cifre da inserire nelle caselle vuote. Tale sequenza ha chiaramente lunghezza m ed è quindi polinomiale in n: inoltre, possiamo facilmente verificare la correttezza del problema proposto dall'editore, riempiendo le caselle vuote con le cifre della sequenza come riportato nel codice seguente.

```
1  VerificaSudoku( sequenza ):        ⟨pre: sequenza di m cifre, con 0 < m ≤ n²⟩
2    casella = PrimaVuota( );
3    FOR (i = 0; i < m; i = i+1) {
4      cifra = sequenza[i];
5      IF (cifra appare in casella.riga) RETURN FALSE;
6      IF (cifra appare in casella.colonna) RETURN FALSE;
7      IF (cifra appare in casella.sotto tabella) RETURN FALSE;
8      Assegna( casella, cifra );
9      casella = SuccVuota(casella);
10   }
11   RETURN TRUE;
```

Notiamo che le tre verifiche alle righe 5-7 possono essere eseguite in circa n passi, per cui l'intero algoritmo di verifica richiede circa $m \times n \leq n^3$ passi, ed è quindi polinomiale. In conclusione, **verificare** che una sequenza di m cifre sia una soluzione di un'istanza del Sudoku può essere fatto in tempo polinomiale mentre, a oggi, nessuno conosce un algoritmo polinomiale per **trovare** una tale sequenza.

Insomma, il problema del Sudoku si trova in uno stato di limbo computazionale nella nostra classificazione della Figura 2.

Tale problema non è un esempio isolato, ma esistono decine di migliaia di problemi simili che ricorrono in situazioni reali, che vanno dall'organizzazione del trasporto a problemi di allocazione ottima di risorse. Questi problemi formano la **classe NP** e sono caratterizzati dall'ammettere particolari sequenze binarie chiamate **certificati polinomiali**: chi ha la soluzione per un'istanza di un problema in NP, può convincerci di ciò fornendo un opportuno certificato che ci permette di verificare, in tempo polinomiale, l'esistenza di una qualche soluzione. Notiamo che chi non ha tale soluzione, può comunque procedere per tentativi in tempo esponenziale, provando a generare (più o meno esplicitamente) tutti i certificati possibili.

Come mostrato nella Figura 2, la classe NP include (non sappiamo se in senso stretto o meno) la classe P in quanto, per ogni problema che ammette un algoritmo polinomiale, possiamo usare tale algoritmo per produrre una soluzione e, quindi, un certificato polinomiale.

Il problema del Sudoku in realtà appartiene alla classe dei **problemi NP-completi** (NPC), che sono stati introdotti indipendentemente all'inizio degli anni '70 da due informatici, lo statunitense/canadese Stephen Cook e il russo Leonid Levin. Tali problemi sono i più difficili da risolvere algoritmicamente all'interno della classe NP, nel senso che se scopriamo un algoritmo polinomiale per un qualsiasi problema NP-completo, allora tutti i problemi in NP sono risolubili in tempo polinomiale (ovvero la classe NP coincide con la classe P). Se invece dimostriamo che uno dei problemi NP-completi è intrattabile (e quindi che la classe NP è diversa dalla classe P), allora risultano intrattabili tutti i problemi in NPC.

I problemi studiati in questo libro si collocano principalmente nella classe NP, di cui forniremo una trattazione rigorosa nell'ultimo capitolo. Per il momento anticipiamo che, in effetti, il concetto di NP-completezza fa riferimento ai soli problemi decisionali (ovvero, problemi per i quali la soluzione è binaria – sì o no): con un piccolo abuso di terminologia, indicheremo nel seguito come NP-completi anche problemi che richiedono la ricerca di una soluzione non binaria e che sono computazionalmente equivalenti a problemi decisionali NP-completi.

I problemi in NP (e quindi quelli NP-completi) influenzano la vita quotidiana più di quanto possa sembrare: come detto, se qualcuno mostrasse che i problemi NP-completi ammettono algoritmi polinomiali, ovvero che P = NP, allora ci sarebbero conseguenze in molte applicazioni di uso comune. Per esempio, diventerebbe possibile indovinare in tempo polinomiale una parola chiave di n simboli scelti in modo casuale, per cui diversi metodi di autenticazione degli utenti basati su parole d'ordine e di crittografia basata su chiave pubblica non sarebbero più sicuri (come il protocollo *secure sockets layer* adoperato dalle banche e dal commercio elettronico per le connessioni sicure nel Web).

Non a caso, nel 2000 è stato messo in palio dal *Clay Mathematics Institute* un premio milionario per chi riuscirà a dimostrare che l'uguaglianza P = NP sia vera o meno (la maggioranza degli esperti congettura che sia P ≠ NP per cui possiamo parlare di apparente intrattabilità): risolvendo uno dei due problemi aperti menzionati finora (Goldbach e NP) è quindi possibile diventare milionari.

Esercizio svolto 1 Dato un insieme `A` di `n` numeri interi positivi a_0, …, a_{n-1} e dato un numero intero `s` positivo, ci chiediamo se esista un sottoinsieme di `A`, la somma dei cui elementi sia uguale a `s`. Progettare un algoritmo basato sulla tecnica del backtrack per risolvere tale problema.

Soluzione Come già osservato nel Paragrafo 4, tutti i possibili sottoinsiemi dell'insieme `A` possono essere elencati facendo uso della funzione `GeneraBinarie`. Possiamo modificare tale funzione in modo da utilizzare la tecnica del backtrack per risolvere il problema dato, interrompendo la generazione di un sottoinsieme nel momento in cui ci si accorga che la somma dei suoi elementi sia uguale a `s` (nel qual caso, possiamo restituire il sottoinsieme generato), oppure sia maggiore di `s` (nel qual caso, non ha senso continuare a estendere il sottoinsieme attuale). Lasciamo allo studente il compito di modificare in modo opportuno il codice di `GeneraBinarie`.

6 Modello RAM e complessità computazionale

La classificazione dei problemi discussa finora e rappresentata graficamente nella Figura 2, fa riferimento al concetto intuitivo di **passo elementare**: diamo ora una specifica più formale di tale concetto, attraverso una breve escursione nella struttura logica di un calcolatore.

L'idea di memorizzare sia i dati che i programmi come sequenze binarie nella memoria del calcolatore è dovuta principalmente al grande e controverso scienziato ungherese John von Neumann[7] negli anni '50, il quale si ispirò alla macchina universale di Turing. I moderni calcolatori mantengono una struttura logica simile a quella introdotta da von Neumann, di cui il modello RAM (*Random Access Machine* o macchina ad accesso diretto) rappresenta un'astrazione: tale modello consiste in un processore di calcolo a cui viene associata una memoria di dimensione illimitata, in grado di contenere sia i dati che il programma da eseguire. Il processore dispone di un'unità centrale di elaborazione e di due registri, ovvero il contatore di programma che indica la prossima istruzione da eseguire e l'accumulatore che consente di eseguire le seguenti istruzioni elementari:[8]

[7] Il saggio L'apprendista stregone di Piergiorgio Odifreddi descrive la personalità di von Neumann.

[8] Notiamo che le istruzioni di un linguaggio ad alto livello come C, C++ e JAVA, possono essere facilmente tradotte in una serie di tali operazioni elementari.

- operazioni aritmetiche: somma, sottrazione, moltiplicazione, divisione;
- operazioni di confronto: minore, maggiore, uguale e così via;
- operazioni logiche: and, or, not e così via;
- operazioni di trasferimento: lettura e scrittura da accumulatore a memoria;
- operazioni di controllo: salti condizionati e non condizionati.

Allo scopo di analizzare le prestazioni delle strutture di dati e degli algoritmi presentati nel libro, seguiamo la convenzione comunemente adottata di assegnare un **costo uniforme** alle suddette operazioni. In particolare, supponiamo che ciascuna di esse richieda un tempo *costante* di esecuzione, indipendente dal numero dei dati memorizzati nel calcolatore. Il costo computazionale dell'esecuzione di un algoritmo, su una specifica istanza, è quindi espresso in termini di **tempo**, ovvero il numero di istruzioni elementari eseguite, e in termini di **spazio**, ovvero il massimo numero di celle di memoria utilizzate durante l'esecuzione (*oltre* a quelle occupate dai dati in ingresso).

Per un dato problema, è noto che esistono infiniti algoritmi che lo risolvono, per cui il progettista si pone la questione di selezionarne il migliore in termini di **complessità in tempo** e/o di **complessità in spazio**. Entrambe le complessità sono espresse in notazione **asintotica** in funzione della dimensione n dei dati in ingresso, ignorando così le costanti moltiplicative e gli ordini inferiori.[9] Ricordiamo che, in base a tale notazione, data una funzione f, abbiamo che:

- $O(f(n))$ denota l'insieme di funzioni g tali che esistono delle costanti c, $n_0 > 0$ per cui vale $g(n) \leq cf(n)$, per ogni $n > n_0$ (l'appartenenza di g viene solitamente indicata con $g(n) = O(f(n))$);
- $\Omega(f(n))$ denota l'insieme di funzioni g tali che esistono delle costanti c, $n_0 > 0$ per cui vale $g(n) \geq cf(n)$, per ogni $n > n_0$ (l'appartenenza di g viene solitamente indicata con $g(n) = \Omega(f(n))$);
- $\Theta(f(n))$ denota l'insieme di funzioni g tali che $g(n) = O(f(n))$ e $g(n) = \Omega(f(n))$ (l'appartenenza di g viene solitamente indicata con $g(n) = \Theta(f(n))$);
- $o(f(n))$ denota l'insieme di funzioni g tali che $\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$ (l'appartenenza di g viene solitamente indicata con $g(n) = o(f(n))$).

[9] Gad Landau usa la seguente metafora: un miliardario rimane tale sia che possegga un miliardo di euro che ne possegga nove, o che possegga anche diversi milioni (le costanti moltiplicative negli ordini di grandezza e gli ordini inferiori scompaiono con la notazione asintotica O, Ω e Θ).

Solitamente, si cerca prima di minimizzare la complessità asintotica in tempo e, a parità di costo temporale, la complessità in spazio: la motivazione è che lo spazio può essere riusato mentre il tempo è irreversibile.[10]

Nella complessità al **caso pessimo** o **peggiore** consideriamo il costo massimo su tutte le possibili istanze di dimensione `n`, mentre nella complessità al **caso medio** consideriamo il costo mediato tra tali istanze. La maggior parte degli algoritmi presentati in questo libro saranno analizzati facendo riferimento al caso pessimo, ma saranno mostrati anche alcuni esempi di valutazione del costo al caso medio.

Diamo ora una piccola guida per valutare al caso pessimo la complessità in tempo di alcuni dei costrutti di programmazione più frequentemente usati nel libro (come ogni buona catalogazione, vi sono le dovute eccezioni che saranno illustrate di volta in volta).

- Le singole operazioni logico-aritmetiche e di assegnamento hanno un costo costante.
- Nel costrutto condizionale

  ```
  IF (guardia) { blocco1 } ELSE { blocco2 }
  ```

  uno solo tra i rami viene eseguito, in base al valore di `guardia`. Non potendo prevedere in generale tale valore e, quindi, quale dei due blocchi sarà eseguito, il costo di tale costrutto è pari a

  ```
  costo(guardia) + max{costo(blocco1), costo(blocco2)}
  ```

- Nel costrutto iterativo

  ```
  FOR (i = 0; i < m; i = i + 1) { corpo }
  ```

  sia t_i il costo dell'esecuzione di `corpo` all'iterazione `i` del ciclo (come vedremo nel libro, non è detto che `corpo` debba avere sempre lo stesso costo a ogni iterazione). Il costo risultante è proporzionale a

  $$\sum_{i=0}^{m-1} t_i$$

  (assumendo che `corpo` non modifichi la variabile `i` e che i costi t_i siano positivi).

[10] In alcune applicazioni, come vedremo, lo spazio è importante quanto il tempo, per cui cercheremo di minimizzare entrambe le complessità con algoritmi più sofisticati.

- Nei costrutti iterativi

```
WHILE (guardia) { corpo }
DO { corpo } WHILE (guardia);
```

  sia `m` il numero di volte in cui `guardia` è soddisfatta. Sia t'_i il costo della sua valutazione all'iterazione `i` del ciclo, e t_i il costo di `corpo` all'iterazione `i`. Poiché `guardia` viene valutata una volta in più rispetto a `corpo` nel primo costrutto e lo stesso numero di volte nel secondo costrutto, in entrambi i casi abbiamo che, assumendo che i costi t'_i e t_i siano positivi, il costo totale è al più

$$\sum_{i=0}^{m} t'_i + \sum_{i=0}^{m-1} t_i$$

  (notiamo che, di solito, la parte difficile rispetto alla valutazione del costo per il ciclo `FOR`, è fornire una stima del valore di `m`).

- Il costo della chiamata a funzione è dato da quello del corpo della funzione stessa più quello dovuto al calcolo degli argomenti passati al momento dell'invocazione (come vedremo, nel caso di funzioni ricorsive, la valutazione del costo sarà effettuata nel libro mediante la risoluzione delle relative equazioni di ricorrenza).
- Infine, il costo di un blocco di istruzioni e costrutti visti sopra è pari alla somma dei costi delle singole istruzioni e dei costrutti, secondo quanto appena discusso.

Per concludere, osserviamo che la valutazione asintotica del costo di un algoritmo serve a identificare algoritmi chiaramente inefficienti *senza* il bisogno di implementarli e sperimentarli. Per gli algoritmi che risultano invece efficienti (da un punto di vista di analisi della loro complessità), occorre tener conto del particolare sistema che intendiamo usare (piattaforma *hardware* e livelli di memoria, sistema operativo, linguaggio adottato, compilatore e così via). Questi aspetti sono volutamente ignorati nel modello RAM per permettere una prima fase di selezione ad alto livello degli algoritmi promettenti, che però necessitano di un'ulteriore indagine sperimentale che dipende anche dall'applicazione che intendiamo realizzare: come ogni modello, anche la RAM non riesce a catturare le mille sfaccettature della realtà.

7 Alla ricerca del miglior algoritmo

Per illustrare i principi di base che guidano la metodologia di progettazione di algoritmi il più efficienti possibile e la loro analisi di complessità in termini di tempo e spazio, consideriamo un problema "giocattolo", ovvero la ricerca del *segmento di somma massima*. Data una sequenza di `n` interi memorizzata in un

array a, un **segmento** è una qualunque sotto-sequenza di elementi consecutivi, a_i, a_{i+1}, …, a_j, dalla posizione i fino alla j. Tale segmento viene indicato con la notazione a[i,j], dove $0 \le i \le j \le n-1$; in tal modo, l'intero array corrisponde ad a[0,n-1]. La *somma* di un segmento a[i,j] è data dalla somma dei suoi componenti, $\texttt{somma(a[i,j])} = \sum_{k=i}^{j} \texttt{a[k]}$. Il problema consiste nell'individuare in a un segmento di somma *massima*, dove a parità di somma viene scelto il segmento più corto.

Notiamo che se a contiene solo elementi positivi, allora il segmento di somma massima coincide necessariamente con l'intero array: il problema diventa interessante se l'array include almeno un elemento negativo. D'altra parte, se a contiene solo elementi negativi, allora il problema si riduce a trovare l'elemento dell'array il cui valore assoluto è minimo: per questo motivo, possiamo supporre che a contenga almeno un elemento positivo (è chiaro che, in questo caso, un segmento di somma massima deve avere gli estremi positivi, in quanto altrimenti potremmo ottenere un segmento avente somma maggiore escludendo un estremo negativo).

Nella prima soluzione proposta, generiamo direttamente tutti i segmenti calcolando la somma massima. Un segmento a[i,j] è univocamente identificato dalla coppia di posizioni, i e j, dei suoi estremi a[i] e a[j]. Generiamo quindi tutte le coppie i e j in cui $0 \le i \le j \le n-1$ e calcoliamo le somme dei relativi segmenti, ottenendo l'algoritmo mostrato nel Codice 1. Il costo di tale algoritmo è $O(n^3)$ tempo: infatti, il corpo dei primi due cicli FOR (dalla riga 5 alla 8) viene eseguito meno di n^2 volte e, al suo interno, il terzo ciclo FOR calcola somma(a[i,j]) eseguendo $j - i + 1$ iterazioni, ciascuna in tempo O(1). D'altra parte l'algoritmo richiede $\Omega(n^3)$ tempo, in quanto la riga 7 è eseguita $\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 1)$ volte, ovvero $\sum_{i=0}^{n-1} \sum_{j=1}^{n-i} j \ge \sum_{i=0}^{n-1} (n - i)^2/2 \ge \sum_{i=0}^{n/2} n^2/8 = \Omega(n^3)$ volte. La complessità in spazio è O(1) in quanto usiamo soltanto un numero costante di variabili di appoggio oltre ai dati in ingresso.

ALVIE **Codice 1** Prima soluzione per il segmento di somma massima.

```
1  SommaMassima1( a ):   ⟨pre: a contiene n elementi di cui almeno uno positivo⟩
2    max = 0;
3    FOR (i = 0; i < n; i = i+1) {
4      FOR (j = i; j < n; j = j+1) {
5        somma = 0;
6        FOR (k = i; k <= j; k = k+1)
7          somma = somma + a[k];
8        IF (somma > max) max = somma;
9      }
10   }
11   RETURN max;
```

Nella seconda soluzione proposta, una volta calcolata `somma(a[i,j-1])`, evitiamo di ripartire da capo per il calcolo di `somma(a[i,j])`. Utilizziamo il fatto che `somma(a[i,j])` = `somma(a[i,j-1])` + `a[j]`, ottenendo il Codice 2. Mantenendo l'invariante che, all'inizio di ogni iterazione del ciclo `FOR` più interno (dalla riga 5 alla 8), la variabile `somma` corrisponde a `somma(a[i,j-1])`, è sufficiente aggiungere `a[j]` per ottenere `somma(a[i,j])`. Il costo in tempo è ora $O(n^2)$ in quanto dettato dai due cicli `FOR`, mentre la complessità in spazio rimane $O(1)$.

ALVIE **Codice 2** Seconda soluzione per il segmento di somma massima.

```
1  SommaMassima2( a ): ⟨pre: a contiene n elementi di cui almeno uno positivo⟩
2    max = 0;
3    FOR (i = 0; i < n; i = i+1) {
4      somma = 0;
5      FOR (j = i; j < n; j = j+1) {
6        somma = somma + a[j];
7        IF (somma > max) max = somma;
8      }
9    }
10   RETURN max;
```

Nella terza e ultima soluzione proposta, sfruttiamo meglio la struttura combinatoria del problema in quanto, a fronte di $O(n^2)$ possibili segmenti, ne possono esistere soltanto $O(n)$ di somma massima, e questi sono disgiunti a seguito della seguente proprietà invariante. Esaminando un segmento di somma massima, `a[i,j]`, notiamo che deve avere lunghezza minima tra i segmenti di pari somma, e deve soddisfare le seguenti due condizioni.

(a) Ogni prefisso di `a[i,j]` ha somma positiva: `somma(a[i,k]) > 0` per ogni $i \le k < j$. Se così non fosse, esisterebbe un valore di `k` tale che

$$\texttt{somma(a[i,j])} = \texttt{somma(a[i,k])} + \texttt{somma(a[k+1,j])} \le \texttt{somma(a[k+1,j])}$$

ottenendo una contraddizione in quanto il segmento `a[k+1,j]` avrebbe somma maggiore o, a parità di somma, sarebbe più corto.

(b) Il segmento `a[i,j]` non può essere esteso a sinistra: `somma(a[k,i-1])` $\le$ `0` per ogni `0` $\le$ `k` $\le$ `i` $-$ `1`. Se così non fosse, esisterebbe una posizione `k` $\le$ `i` $-$ `1` per cui

$$\texttt{somma(a[k,j])} = \texttt{somma(a[k,i-1])} + \texttt{somma(a[i,j])} > \texttt{somma(a[i,j])}$$

ottenendo una contraddizione in quanto il segmento `a[k,j]` avrebbe somma maggiore di quello con somma massima.

Sfruttiamo le proprietà (a) e (b) durante la scansione dell'array a, come mostrato nel Codice 3. In particolare, la riga 6 corrisponde all'applicazione della proprietà (a), che ci assicura che possiamo estendere a destra il segmento corrente. La riga 8, invece, corrisponde all'applicazione della proprietà (b), in base alla quale possiamo scartare il segmento corrente e iniziare a considerare un nuovo segmento disgiunto da quelli esaminati fino a quel punto.

ALVIE **Codice 3** Terza soluzione per il segmento di somma massima.

```
1  SommaMassima3( a ):   ⟨pre: a contiene n elementi di cui almeno uno positivo⟩
2    max = 0;
3    somma = max;
4    FOR (j = 0; j < n; j = j+1) {
5      IF (somma > 0) {
6        somma = somma + a[j];
7      } ELSE {
8        somma = a[j];
9      }
10     IF (somma > max) max = somma;
11   }
12   RETURN max;
```

ESEMPIO 4

Consideriamo la sequenza 31, –41, 59, 26, –53, 58, 97, –93, –95, 14, –10, 20. Inizialmente somma e max sono uguali a 0. Analizzando il primo elemento della sequenza, sia somma che max vengono modificati e posti uguale a 31. Poiché somma è positivo, il suo valore viene aggiornato a 31 – 41 = –10, nel momento in cui il secondo elemento viene analizzato: in questo caso, chiaramente, max rimane invariato. Analizzando il terzo elemento, l'algoritmo dà inizio a un nuovo segmento, in quanto il valore di somma è negativo. Questo segmento si estende fino al settimo elemento dando luogo a un valore di somma e di max pari a 187. I successivi due elementi, essendo negativi, non possono aumentare il valore di max e rendono il valore di somma negativo, così che, al momento di analizzare il terzultimo elemento, l'algoritmo dà inizio a un nuovo segmento che si estende fino all'ultimo elemento producendo un valore di somma pari a 24, ma non modificando il valore di max. L'algoritmo termina quindi restituendo il valore 187.

Il costo di quest'ultima soluzione è $O(n)$ in quanto c'è un solo ciclo FOR, il cui corpo richiede $O(1)$ passi a ogni iterazione. Lo spazio aggiuntivo richiesto è rimasto di $O(1)$ locazioni di memoria. In conclusione, partendo dalla prima soluzione, abbiamo ridotto la complessità in tempo da $O(n^3)$ a $O(n^2)$ con la seconda soluzione, per poi passare a $O(n)$ con la terza. Invitiamo il lettore a eseguire sul

calcolatore le tre soluzioni proposte per rendersi conto che la differenza di complessità non è confinata a uno studio puramente teorico, ma molto spesso incide sulle prestazioni reali.

Notiamo che l'algoritmo per la terza soluzione ha una complessità asintotica *ottima* sia in termini di spazio che di tempo. Nel caso dello spazio, ogni algoritmo deve usare almeno un numero costante di locazioni per le variabili di appoggio. Per il tempo, ogni algoritmo per il problema deve leggere tutti gli `n` elementi dell'array `a` perché, se così non fosse, avremmo almeno un elemento, `a[r]`, non letto: in tal caso, potremmo invalidare la soluzione trovata assegnando un valore opportuno ad `a[r]`. Quindi *ogni* algoritmo risolutore per il problema del segmento di somma massima deve leggere *tutti* gli elementi e pertanto richiede un tempo pari a $\Omega(n)$. Ne consegue che la terza soluzione è ottima asintoticamente. In generale, pur essendoci un numero infinito di algoritmi risolutori per un dato problema, possiamo derivare degli argomenti formali per dimostrare che l'algoritmo proposto è tra i migliori dal punto di vista della complessità computazionale.

7.1 Limiti superiori e inferiori

Il problema del segmento di somma massima esemplifica l'approccio concettuale adottato nello studio degli algoritmi. Per un dato problema computazionale Π, consideriamo un qualunque algoritmo `A` di risoluzione. Se `A` richiede `t(n)` tempo per risolvere una generica istanza di Π di dimensione `n`, diremo che $O(t(n))$ è un **limite superiore** alla complessità in tempo del problema Π. Lo scopo del progettista è quello di riuscire a trovare l'algoritmo `A` con il migliore tempo `t(n)` possibile.

A tal fine, quando riusciamo a dimostrare con argomentazioni combinatorie che *qualunque* algoritmo `A'` richiede *almeno* tempo `f(n)` per risolvere Π su un'istanza generica di dimensione `n`, asintoticamente per infiniti valori di `n`, diremo che $\Omega(f(n))$ è un **limite inferiore** alla complessità in tempo del problema Π. In tal caso, nessun algoritmo può richiedere asintoticamente meno di $O(f(n))$ tempo per risolvere Π.

Ne deriva che l'algoritmo `A` è **ottimo** se $t(n) = O(f(n))$, ovvero se la complessità in tempo di `A` corrisponde dal punto di vista asintotico al limite inferiore di Π. Ciò ci permette di stabilire che la **complessità computazionale del problema** è $\Theta(f(n))$. Notiamo che spesso la complessità computazionale di un problema combinatorio Π viene confusa con quella di un suo algoritmo risolutore `A`: in realtà ciò è corretto se e solo se `A` è ottimo.

Nel problema del segmento di somma massima, abbiamo mostrato che la terza soluzione richiede tempo $O(n)$. Quindi il limite superiore del problema è $O(n)$. Inoltre, abbiamo mostrato che ogni algoritmo di risoluzione richiede $\Omega(n)$ tempo, fornendo un limite inferiore. Ne deriva che la complessità del problema è $\Theta(n)$ e che la terza soluzione è un algoritmo ottimo.

Per quanto riguarda la complessità in spazio, `s(n)`, possiamo procedere analogamente al tempo nella definizione di limite superiore e inferiore, nonché di ottimalità. Da ricordare che lo spazio `s(n)` misura il numero di locazioni di memoria necessarie a risolvere il problema Π, *oltre* a quelle richieste dai dati in ingresso. Per esempio, gli algoritmi **in loco** sono caratterizzati dall'usare soltanto spazio $s(n) = O(1)$, come accade per il problema del segmento di somma massima.

8 Esercizi

1. L'ultimo teorema di Fermat afferma che l'equazione $x^n + y^n = z^n$ non ha soluzioni intere per ogni coppia di interi $x, y > 0$ e per ogni intero $n > 2$. Dimostrare che se esistesse il programma `Termina`, allora potremmo provare o confutare l'ultimo teorema di Fermat.

2. Dimostrare che non esiste un programma `TerminaZero`, il quale, preso un programma `A`, restituisce (in tempo finito) un valore di verità per indicare che `A` termina o meno quando viene eseguito con input `0`.

3. Per ogni $i \geq 1$, l'i-esimo numero di Fibonacci `F(i)` è definito nel modo seguente:

$$F(i) = \begin{cases} 1 & \text{se } i = 1, 2 \\ F(i-1) + F(i-2) & \text{altrimenti} \end{cases}$$

   Sulla base di tale definizione, scrivere una funzione ricorsiva `Fibonacci` che calcoli il valore `F(i)`. Valutare il numero di passi che tale funzione esegue al variare del numero `i` e, usando il fatto che $F(i) = \left\lfloor \frac{\phi^i}{\sqrt{5}} + \frac{1}{2} \right\rfloor \geq \frac{\phi^i}{\sqrt{5}} - 1$ dove $\phi = \frac{1+\sqrt{5}}{2}$ è il *rapporto aureo*, dimostrare che tale numero cresce esponenzialmente in `i`. Descrivere poi un algoritmo iterativo polinomiale in `i`.

4. Progettate un algoritmo ricorsivo per generare tutti i sottoinsiemi di taglia `n` ottenibili da un insieme di `m` elementi, in cui il numero di chiamate ricorsive effettuate è proporzionale al numero di sottoinsiemi generati.

5. Descrivere un'istanza del Sudoku di dimensione 4×4 contenente quattro celle occupate tale che l'algoritmo di backtrack trovi una soluzione senza mai tornare indietro sulle proprie scelte.

6. Descrivere un algoritmo di backtrack per la risoluzione del problema delle `n` regine che può essere descritto nel modo seguente: `n` regine devono essere poste su una scacchiera di dimensione $n \times n$ in modo tale che nessuna regina possa mangiarne un'altra (ricordiamo che una regina può mangiare un'altra regina se si trova sulla stessa riga, sulla stessa colonna o sulla stessa diagonale).

7. In alcuni casi, per l'analisi di algoritmi operanti su numeri, è necessario fare a meno dell'ipotesi che il costo di un'operazione sia costante: in particolare ciò risulta necessario per rendere il costo di esecuzione di un'operazione aritmetica dipendente dal valore dei relativi argomenti. Una tipica ipotesi, in tal caso, è quella di considerare il costo di un'addizione $n_1 + n_2$ tra due interi proporzionale alla lunghezza della codifica del più grande tra i due, e quindi a $\log \max\{n_1, n_2\}$. Valutare, sotto tale ipotesi, il conseguente costo di una moltiplicazione $n_1 \times n_2$ effettuata mediante il normale procedimento imparato alla scuola elementare.

8. Per ciascuna delle seguenti funzioni, indicare la più piccola classe $O(\cdot)$ che la contiene: $f_1(n) = 6\sqrt{n} + 2n^3 + 127n^2 + 1024n$, $f_2(n) = 3n^2 \log n + 12 \log^5 n + 6n$, $f_3(n) = 4 \log^2 n + \sqrt[3]{x^2}$, $f_4(n) = 1.02^{n/100} + 2048n^7$ e $f_5(n) = \sqrt{\log(2^{n^2})} + 36\sqrt{n}$.

9. Dimostrare che se $f(n) = O(g(n))$ e $g(n) = O(h(n))$, allora $f(n) = O(h(n))$.

10. Mostrare che, se un algoritmo per la risoluzione del problema del segmento di somma massima non legge un elemento `a[r]`, è sempre possibile assegnare ad `a[r]` un valore tale da invalidare la soluzione calcolata dall'algoritmo.