Capitolo 3

Algoritmi e complessità

Una volta stabilita la relazione tra il mondo dei problemi, o delle funzioni associate a essi, e il mondo degli algoritmi, vediamo lungo quali direttrici principali si sviluppano questi ultimi, come si descrivono in pratica e come se ne determina la qualità.

Nel capitolo precedente abbiamo appreso che dobbiamo rivolgerci a un insieme ristretto di problemi: potranno essere problemi di decisione, quindi a risultato binario, o problemi più generali, ma in ogni caso dovranno ammettere un algoritmo di soluzione. E tra questi problemi scopriremo un'altra linea che li divide in due famiglie drasticamente separate: problemi *trattabili* che ammettono algoritmi di soluzione praticamente eseguibili, o *intrattabili* se tutti gli algoritmi possibili, o quanto meno noti, sono così inefficienti da impedirne una pratica esecuzione su qualsiasi calcolatore.

L'avvento dei calcolatori è stato accompagnato dallo sviluppo di *linguaggi di programmazione* in cui ogni dettaglio deve essere perfettamente specificato perché non è previsto che la macchina prenda decisioni autonome. In conseguenza, benché gli algoritmi possano essere descritti utilizzando un qualunque formalismo univocamente interpretabile, è consuetudine descriverli in un linguaggio di tipo progammativo anche se eventualmente non coincidente con alcuno di essi. Seguiremo questa via fidando che gli elementi di base della programmazione siano già noti al lettore o siano direttamente comprensibili dal contesto; e inseriremo comunque qualche spiegazione nel testo per accertarci della corretta interpretazione dei costrutti più complessi. Questo modo di procedere ci consentirà di descrivere tutte le operazioni in modo preciso e non ambiguo, bene adattandosi alla natura eminentemente matematica dei problemi che dovremo trattare: e per iniziare descriveremo due algoritmi antichissimi, ma ancora molto utili oggi, che nei documenti originali furono redatti rispettivamente in caratteri ieratici egizi e greci, assai diversi dai nostri.

3.1 Paradigmi algoritmici

Verso il 1650 a.C. lo scriba egiziano Ahmes redasse un papiro di argomento matematico tratto, per sua dichiarazione, da un esemplare del Regno Medio. Questo straordinario documento, conservato quasi per intero nel British Museum, contiene un gran numero di definizioni, problemi e regole di calcolo tra cui vogliamo ricordare un algoritmo per eseguire la moltiplicazione tra interi positivi. Detti a e b i due fattori, e p il loro prodotto, l'algoritmo egizio si può descrivere nella forma seguente:

`Function Molt(`a, b`):`
  $p \leftarrow 0$;
  while $a \geq 1$ **do**
    if a è dispari **then** $p \leftarrow p + b$;
    $a \leftarrow \lfloor a/2 \rfloor$;
    $b \leftarrow b \times 2$;
  return p.

Perché non vi siano dubbi sul formalismo adottato ricordiamo che:

- Con la parola `Function` si intende un'unità di programma che assume un valore calcolato al suo interno (p nell'esempio) e rilasciato attraverso il comando **return**, il quale ordina anche l'arresto del calcolo nell'unità stessa. Alla parola `Function` seguono il *nome* dell'unità di programma e, tra parentesi, i *parametri* su cui essa è chiamata a operare: a ogni impiego verranno comunicati all'unità i valori di tali parametri.

- La freccia comanda l'assegnazione, alla *variabile* che appare dalla parte della punta, del *valore* di quanto appare dalla parte della cocca ove può essere eseguito un calcolo il cui risultato sarà oggetto dell'assegnazione. Nell'esempio la frase $a \leftarrow \lfloor a/2 \rfloor$ ordina di dividere per due il valore corrente di a, prendere la parte intera, assegnare il risultato ad a che assume così un nuovo valore: si noti che la stessa variabile a avrà valore diverso a seconda della parte della freccia in cui si trova.

- Il costrutto **while** `C` **do** `S` ordina la ripetizione della sequenza `S` di passi (individuati dalla stessa indentatura) fintanto che è verificata la condizione `C`; ciò implica che nella sequenza `S` venga cambiato qualche elemento che appare in `C` (nell'esempio il valore di a), altrimenti il ciclo sarebbe ripetuto in eterno. Il costrutto **if** `C` **then** `S` dovrebbe avere significato ovvio.

- Il paradigma algoritmico che prevede l'indicazione esplicita di tutte le iterazioni del calcolo è detto *iterativo*. L'algoritmo dell'esempio è iterativo: la ripetizione dei passi computazionali è indicata esplicitamente nel costrutto **while**.

- Il nuovo programma:

  $r \leftarrow a +$ `Molt`(b, c)

  contiene una *chiamata* della funzione `Molt` il cui risultato viene sommato al valore di a per ottenere r. Eseguito per i valori $a = 5$, $b = 7$, $c = 3$ il programma dà come risultato $r = 5 + 7 \times 3 = 26$.

La correttezza dell'algoritmo `Molt` si dimostra immediatamente notando che $a \times b = a/2 \times 2b$ se a è pari, e $a \times b = (a-1) \times b + b = \lfloor a/2 \rfloor \times 2b + b$ se a è dispari. Seguiamone l'esecuzione per i valori $a = 37$ e $b = 23$, quindi $p = 37 \times 23 = 851$, prima di commentarne la struttura:

$p + b$	p	a	b
	0	37	23
0 + 23	23	18	46
	23	9	92
23 + 92	115	4	184
	115	2	368
	115	1	736
115 + 736	851	0	

È chiaro che questo algoritmo è completamente diverso da quello impiegato usualmente da noi. Il calcolo procede per addizioni, nonché per dimezzamenti e raddoppiamenti dei fattori, e questo induce a pensare che la sua ideazione si debba all'uso di abaci che permettevano di eseguire facilmente queste tre operazioni.[1] Rispetto al nostro, l'algoritmo si impone per la totale indipendenza dal metodo di rappresentazione dei numeri: gli antichi egiziani, per esempio, impiegavano una notazione decimale ma non conoscevano lo zero. Il confronto di efficienza tra i due algoritmi è rimandato al prossimo paragrafo; per il momento notiamo che se l'algoritmo egizio è applicato a

[1] Fino a tempi molto recenti l'algoritmo era in uso in Russia e nei paesi asiatici confinanti ove i calcoli di ogni giorno si eseguivano correntemente col pallottoliere, e invero è l'unico metodo pratico per eseguire la moltiplicazione con tale strumento. Per questo motivo il metodo è spesso indicato nei libri come *moltiplicazione del contadino russo*.

due numeri binari, i test e i calcoli aritmetici sono gli stessi eseguiti dai calcolatori nella loro realizzazione più semplice della moltiplicazione.[2]

Il secondo algoritmo che consideriamo ha "solo" duemilatrecento anni, poiché apparve negli *Elementi* di Euclide. Dopo aver definito un numero primo (*protos arithmós*) come un numero "misurato" (cioè diviso) solo dall'unità, e aver definito in conseguenza i numeri primi tra loro, Euclide enuncia la seguente proposizione (Libro VII, Prop. II):

Dati due numeri non primi tra loro trovare la loro più grande misura comune.

È il problema della determinazione del *massimo comun divisore*, che studieremo ora per la sua grande importanza nell'aritmetica in genere e nella crittografia in particolare. All'enunciato Euclide fa seguire la descrizione del metodo di calcolo, tanto efficiente e geniale da essere impiegato ancor oggi adattandolo al nostro formalismo.[3] Dati due interi $a > 0$, $b \geq 0$, con $a \geq b$, l'algoritmo di calcolo del massimo comun divisore *mcd(a,b)* è il seguente:

```
Function Euclid(a, b):
    if b = 0
        then return a
        else return Euclid(b, a mod b).
```

Il formalismo contiene importanti novità:

- Il costrutto **if** `C` **then** `S` **else** `T` ordina l'esecuzione del passo (o sequenza di passi) `S` oppure `T` a seconda che la condizione `C` sia verificata o meno.
- La funzione contiene una *chiamata ricorsiva*, cioè a sé stessa, nella quale i parametri hanno i valori di b e di a `mod` b (ricordiamo che a `mod` b indica il resto della divisione intera tra a e b). Il calcolo della funzione chiamante invoca cioè l'esecuzione di un calcolo uguale su valori diversi dei parametri. La difficoltà pare solo rimandata, e in effetti una struttura di calcolo così concepita è accettabile solo se corredata di una clausola di chiusura che stabilisca come eseguire il calcolo per valori limite dei parametri: nell'esempio il verificarsi della condizione $b = 0$ e corrispondente esecuzione della parte **then** del programma.

[2] La frase: **if** *a è dispari* **then** $p \leftarrow p + b$ corrisponde all'operazione in hardware: se il bit meno significativo del moltiplicatore a è 1, addiziona il moltiplicando b al prodotto parziale p. Le frasi: $a \leftarrow \lfloor a/2 \rfloor$ e $b \leftarrow b \times 2$ corrispondono agli *shift* destro e sinistro dei due fattori.

[3] In particolare è chiaro che Euclide, pur senza farne esplicito riferimento, trattava i numeri come le corrispondenti misure di segmenti: un segmento ne "misurava" un'altro se era in esso contenuto un numero esatto di volte. La sua formulazione dell'algoritmo è basata su questa impostazione: apparentemente Euclide non si curò di esaminarne la straordinaria efficienza poiché considerava l'aritmetica una scienza puramente speculativa.

- Nell'esempio la chiamata ricorsiva costituisce l'ultima frase della funzione chiamante; se ciò non si verifica, il calcolo di questa deve essere sospeso in attesa che sia completato il calcolo della funzione chiamata, e successivamente portato a termine: ciò genera un gran numero di esecuzioni, tutte vive e interrotte una dentro l'altra in attesa di essere completate a partire dalla più "interna".

- Il paradigma algoritmico che prevede l'indicazione di alcune iterazioni del calcolo attraverso chiamate ricorsive è detto a sua volta *ricorsivo*.

Per dimostrare la correttezza dell'algoritmo `Euclid` è sufficiente dimostrare che *mcd(a,b) = mcd(b,* a `mod` b*)*. A tale scopo, ponendo m =*mcd(a,b)*, avremo $a = \alpha m$, $b = \beta m$, con α e β interi e primi tra loro. Possiamo anche porre: $a = q \cdot b + a$ `mod` b ove q è un quoziente intero; da cui: a `mod` $b = (\alpha - q\beta)m = \gamma m$ con γ intero. Dunque m è un divisore anche di a `mod` b. Ora è facile dimostrare che, poiché α e β sono primi tra loro, lo sono anche β e γ, e quindi il massimo comun divisore tra b e a `mod` b è proprio m.

Seguiamo infine l'esecuzione dell'algoritmo per i valori $a = 306$ e $b = 135$. Troveremo in pochi passi $mcd(306, 135) = 9$.

a	b	a `mod` b	*risultato finale*
306	135	36	
135	36	27	
36	27	9	
27	9	0	
9	0		9

I due problemi studiati, e i loro algoritmi di soluzione, sono in genere detti *numerici* poiché i dati sono numeri, in contrapposizione ai problemi e algoritmi *combinatori* in cui più genericamente si richiede di operare su sequenze. Per quanto abbiamo discusso nel capitolo 2 questa distinzione è molto superficiale e al limite illegittima, ma a volte è utile. Vediamo un importante esempio di problema combinatorio.

Dobbiamo ricercare un elemento k in un insieme I: è il problema decisionale $\mathcal{P}_{ric}(k, I)$ già discusso nel paragrafo 2.3, che ora affrontiamo in una forma più generale chiedendo, nel caso che k sia un elemento di I, di indicare anche come "accedere" a tale elemento. Per far questo occorrono alcune precisazioni. I dati, cioè k e gli elementi di I, sono sequenze arbitrarie ma nella pratica hanno spesso una lunghezza massima prefissata: per esempio rappresentano nomi di persone, per ciascuno dei quali è assegnato un numero massimo di caratteri alfabetici. Possiamo allora rappresentare gli elementi di I in un *vettore* A le cui celle sono numerate da 1 a $|A|$; con $A[j]$ si indica l'elemento in posizione j, e se k appartiene all'insieme si deve fornire come risultato

il valore di j per cui $k = A[j]$. Consideriamo il confronto tra due elementi come operazione elementare senza entrare nel merito di come venga eseguita. Il risultato del confronto è binario nella sua forma più semplice, e indica se i due elementi confrontati sono uguali o diversi. Più ambizioso, ma legittimo, è immaginare che tra i dati esista una relazione d'ordine totale e il risultato del confronto sia ternario e indichi se due elementi sono uguali, o se uno precede o segue l'altro secondo la relazione d'ordine. Per indicare questo risultato impiegheremo i simboli aritmetici: $=$, $<$, $>$.[4] Possiamo ora presentare un primo algoritmo di ricerca:

```
Function Ricerca_Sequenziale(k, A):
    for j ← 1 to |A| do
        if k = A[j] then return j;
    return 0.
```

Il formalismo contiene una novità:

- Il costrutto **for** `C` **do** `S` ordina l'esecuzione del passo (o sequenza di passi) `S` per ogni valore intero di un *indice* che varia entro i limiti specificati nel comando `C`. Nell'esempio si comanda all'indice j di assumere in successione tutti i valori interi da 1 a $|A|$: l'algoritmo è quindi iterativo.

Il risultato della funzione `Ricerca_Sequenziale` è un intero: 0 se $k \notin I$, j compreso tra 1 e $|A|$ se $k \in I$. Potrebbe apparire inutile, nel secondo caso, indicare la posizione di k in A poiché in fondo il valore dell'elemento trovato è già noto; ma in realtà l'algoritmo si impiega in genere per la ricerca di *chiavi*, cioè nomi cui sono associate altre informazioni. Esempi immediati sono quelli della ricerca di un nome (la chiave) k nell'elenco del telefono, o di una parola in un dizionario: A rappresenta l'elenco o il dizionario, la risposta $j > 0$ indica la posizione nella quale, assieme a k, si trova il numero telefonico o la definizione cercata. La funzione `Ricerca_Sequenziale` scandisce tutti gli elementi di A fino a incontrare quello cercato: il metodo usualmente svolto a mano è molto diverso, ma per essere eseguito richiede che gli elementi del vettore siano disposti in ordine alfabetico. Vediamo dunque come si possa formalizzare l'algoritmo su un vettore ordinato.

Le operazioni eseguite dall'uomo fanno implicitamente uso di alcune proprietà statistiche delle chiavi elencate: se per esempio l'ordine è alfabetico, una chiave che comincia per d sarà cercata verso l'inizio dell'elenco, salvo poi saltare indietro di un

[4] Nella pratica le sequenze sono binarie e nel confronto vengono interpretate come numeri, sicché la relazione d'ordine è la normale relazione di maggioranza tra interi. Impiegando il codice internazionale ASCII per rappresentare i caratteri alfabetici, due parole ordinate alfabeticamente lo sono anche se interpretate come numeri.

gruppo di pagine se si è raggiunta la lettera f; il meccanismo è via via applicato a salti sempre più brevi, finché si leggono alcune chiavi in sequenza attorno a quella cercata. Questo metodo non è facile da formalizzare per un calcolatore, che in compenso può calcolare facilmente la posizione centrale nell'elenco se questo è memorizzato in un vettore. Il nuovo algoritmo, detto di *ricerca binaria*, impiega tale calcolo per la ricerca di k in una sezione arbitraria di A compresa tra le posizioni i e j, con $1 \leq i \leq j \leq |A|$ (se si raggiunge la condizione $i > j$ l'algoritmo termina dichiarando che k non è presente in A).

`Function Ricerca_Binaria`(k, A, i, j):
if $i > j$ **then return** 0;
$m \leftarrow \lfloor \frac{i+j}{2} \rfloor$;
if $k = A(m)$ **then return** m;
if $k < A(m)$ **then return** `Ricerca_Binaria`$(k, A, i, m-1)$;
if $k > A(m)$ **then return** `Ricerca_Binaria`$(k, A, m+1, j)$.

Sull'impiego di questa funzione è necessario qualche commento:

- La funzione è definita tra estremi arbitrari i, j per poter essere richiamata ricorsivamente al suo interno su sezioni diverse del vettore A. Per cercare k nell'intero vettore si impiegherà la chiamata:

  `Ricerca_Binaria`$(k, A, 1, |A|)$

  che innesca il meccanismo di calcolo sui sottovettori.

- Nella sezione di A compresa tra i e j, il valore $m = \lfloor \frac{i+j}{2} \rfloor$ corrisponde alla posizione scelta come "centrale", che è strettamente al centro solo se il numero di celle della sezione è dispari.

La correttezza dell'algoritmo `Ricerca_Binaria` può essere facilmente dimostrata per induzione. Seguiamone il funzionamento sulla ricerca dell'elemento $k = 48$ nel seguente vettore ordinato A che contiene quattordici interi:

posizione:	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A:	12	17	18	21	23	25	**30**	35	48	50	51	58	65	70
	12	17	18	21	23	25	30	35	48	50	**51**	58	65	70
	12	17	18	21	23	25	30	35	**48**	50	51	58	65	70

Le posizioni esaminate sono 7, 11, 9 (elementi 30, 51, 48) e qui l'algoritmo si arresta trasmettendo il valore 9 dell'indice ove si trova l'elemento cercato. La ricerca di $k = 24$

scandisce le posizioni 7, 3, 5, 6: a questo punto l'algoritmo esegue la chiamata ricorsiva `Ricerca_Binaria`$(16, A, 6, 5)$ (con $i > j$) e qui si arresta trasmettendo il valore 0 che indica l'assenza dell'elemento 24. In ogni caso l'algoritmo esamina pochi elementi del vettore e appare sotto questo aspetto migliore di `Ricerca_Sequenziale`: lo studio di questo miglioramento, che come vedremo cresce grandemente con la lunghezza del vettore, è argomento del prossimo paragrafo.

3.2 Complessità computazionale

Per risolvere un problema siamo in genere interessati a scoprire l'algoritmo "più efficiente" tra tutti i possibili indipendentemente da altre proprietà. Per qualificare questa affermazione dobbiamo chiarire su quali parametri valutare l'efficienza, che normalmente coincide con l'ottimizzazione di impiego delle risorse disponibili. Impiegando un calcolatore le risorse principali sono la memoria e il tempo necessari a eseguire il calcolo: la richiesta di ridurre il tempo è in genere prevalente, al punto che il risparmio di memoria è trascurato a meno che non sia esplicitamente richiesto. "Efficiente" dunque vuol dire veloce, e come vedremo vi sono argomenti molto importanti che avvalorano questa posizione.

È bene chiarire che non misureremo il tempo in secondi, il che non avrebbe alcun senso se non specificando ogni dettaglio del calcolatore e del corredo di software utilizzati, ma attraverso una funzione matematica che assume valori approssimativamente proporzionali al tempo reale: questa funzione misura la *complessità computazionale in tempo* (o semplicemente *complessità*) di un algoritmo. Ciò comporta una certa grossolanità di giudizio ma permette di confrontare l'efficienza di algoritmi diversi in modo molto semplice.

Dobbiamo anzitutto stabilire quali sono le variabili indipendenti su cui operare. In genere la compessità si valuta in funzione di un singolo parametro n detto *dimensione dell'ingresso* o *dimensione dei dati*, che indica la lunghezza della sequenza che specifica il valore dei dati per la particolare istanza del problema che l'algoritmo è chiamato a risolvere. Per esempio se l'algoritmo `Molt`(a, b) del paragrafo precedente è applicato all'istanza di moltiplicazione tra $a = 351$ e $b = 66$ possiamo porre $n = 5$ poiché tante sono le cifre (decimali) con cui si rappresentano i dati. E qui sorge apparentemente un problema poiché il valore indicato dipende dalla *base di numerazione*, cioè dal metodo di rappresentazione scelto. Dobbiamo approfondire questo punto prima di procedere.

Cambiando il metodo di rappresentazione, ma rimanendo nell'ambito di rappresentazioni efficienti nel senso tecnico della parola (paragrafo2.1), le lunghezze delle diverse sequenze che rappresentano uno stesso dato sono *proporzionali tra loro* poiché espresse da funzioni logaritmiche in cui cambia solo la base, ovvero la cardinalità dell'alfabeto scelto. Infatti la funzione logaritmo gode della proprietà:

$$\log_r x = \log_r s \cdot \log_s x \tag{3.1}$$

che mostra come si possa passare tra due basi arbitrarie r, s moltiplicando la funzione per la quantità $\log_r s$, che è una *costante* per r e s costanti (in particolare indipendenti da x). Per esempio un intero N è rappresentato con $\lceil \log_{10} N \rceil$ cifre impiegando una numerazione in base 10. Passando da questa base alla base 2 il numero di cifre necessario viene moltiplicato per $\log_2 10 \simeq 3.32$. In conclusione affinché la complessità di un algoritmo sia indipendente dal metodo di rappresentazione dei dati dovremo limitarci a valutarla in *ordine di grandezza*, prendendo come dimensione n di un problema un valore proporzionale alla lunghezza della sequenza di ingresso.

Nello studio degli algoritmi i tre ordini di grandezza Θ, O e Ω hanno rilevanza particolare. Ne ricordiamo la definizione matematica limitata al caso in cui tutte le funzioni in gioco siano non negative (poiché rappresentano complessità di algoritmi) e siano definite per valori non negativi di una sola variabile n (la dimensione del problema). Gli ordini di grandezza sono insiemi infiniti che comprendono tutte le funzioni che hanno un dato comportamento rispetto a una funzione assegnata $g(n)$:

- $\Theta(g(n))$ è l'insieme di tutte le funzioni $f(n)$ per cui esistono tre costanti positive c_1, c_2, n_0 tali che $c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ per ogni $n \geq n_0$. Per ogni funzione $f(n)$ di questo insieme si scrive, con notazione impropria perché non insiemistica: $f(n) = \Theta(g(n))$, e si legge: $f(n)$ è di ordine *teta* $g(n)$ (una simile terminologia si usa anche per gli altri ordini). Questo significa che, al crescere di n oltre n_0, la funzione $f(n)$ è compresa in una fascia delimitata da $c_1 g(n)$ e $c_2 g(n)$: ovvero che la $f(n)$ ha lo stesso comportamento della $g(n)$ a parte una costante moltiplicativa. Per esempio $3n^2 - 2n + 1 = \Theta(n^2)$ poiché, scegliendo $c_1 = 1$, $c_2 = 3$, $n_0 = 2$ si ha: $n^2 < 3n^2 - 2n + 1 < 3n^2$ per ogni $n \geq 2$.

- $O(g(n))$ è l'insieme di tutte le funzioni $f(n)$ per cui esistono due costanti positive c, n_0 tali che $f(n) \leq c\, g(n)$ per ogni $n \geq n_0$. Questo significa che, al crescere di n oltre n_0, il comportamento della funzione $f(n)$ è limitato superiormente da quello della $g(n)$ a parte una costante moltiplicativa. Per esempio si ha: $3n^2 - 2n + 1 = O(n^2)$, ma anche $3n^2 - 2n + 1 = O(n^3)$.

- $\Omega(g(n))$ è l'insieme di tutte le funzioni $f(n)$ per cui esistono due costanti positive c, n_0 tali che $c\, g(n) \leq f(n)$ per ogni $n \geq n_0$. Questo significa che, al crescere di n oltre n_0, il comportamento della funzione $f(n)$ è limitato inferiormente da quello della $g(n)$ a parte una costante moltiplicativa. Per esempio si ha: $3n^2 - 2n + 1 = \Omega(n^2)$, ma anche $3n^2 - 2n + 1 = \Omega(n^{1.5})$.

- Si noti che $f(n) = \Theta(g(n))$ se e solo se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.

Per valutare l'ordine di grandezza di una funzione composta da una somma di più termini si deve considerare solo il termine che va all'infinito più rapidamente al crescere di n (o termine di grado massimo se la funzione è un polinomio), trascurando gli altri. Similmente si devono trascurare le costanti moltiplicative assorbite dalla definizione di ordine. Così nello studio della funzione $3n^2 - 2n + 1$ l'unico elemento che conta è n^2, sia che si consideri l'ordine Θ, o O, o Ω.

Una considerazione particolare merita la funzione logaritmo che apparirà di continuo nei nostri calcoli. Poiché i logaritmi di uguale argomento ma di diversa base sono tutti proporzionali tra loro (relazione 3.1), la base si ignora entro gli ordini di grandezza poiché ha solo l'effetto di una costante moltiplicativa. Si scrive per esempio: $3(\log_2 n)^3 = \Theta(\log^3 n)$. Ricordiamo inoltre che il logaritmo di n cresce più lentamente di n elevato a qualsiasi esponente positivo, cioè per qualunque $b > 1$ e $\epsilon > 0$ si ha: $\log_b n = O(n^\epsilon)$, $n^\epsilon = \Omega(\log n)$ e $\log_b n \neq \Theta(n^\epsilon)$.

La funzione di complessità (in tempo) è in genere indicata con $T(n)$. Essa viene calcolata in ordine di grandezza non solo per affrancarla dal metodo di rappresentazione dei dati, ma anche per renderla indipendente dalle caratteristiche del sistema di calcolo. Si desidera cioè giungere a una valutazione dell'efficienza degli algoritmi che abbia per quanto possibile validità generale, con lo scopo di poter confrontare e scegliere tra algoritmi diversi, o valutare se un algoritmo è del tutto proponibile. Vediamo cosa si può dire degli algoritmi presentati nel paragrafo precedente.

Per l'algoritmo `Molt(`a, b`)` il calcolo non è semplicissimo. La dimensione dell'ingresso è $n = \Theta(\log a + \log b)$. Il prodotto finale $a \times b$ si rappresenta con $\Theta(\log ab) = \Theta(\log a + \log b)$ cifre, dunque ha la stessa dimensione dell'ingresso. A parte l'operazione iniziale che richiede tempo costante, l'algoritmo ripete un ciclo **while** in cui si eseguono tre operazioni aritmetiche di addizione e *shift* che richiedono complessivamente tempo $\Theta(\log a + \log b)$ perché eseguite su dati di tali dimensioni. Notiamo ora che a ogni ripetizione del ciclo il valore di a vieve dimezzato fino a raggiungere il valore 1, quindi il ciclo è ripetuto $\lfloor \log_2 a \rfloor = \Theta(\log a)$ volte.[5] La complessità di `Molt(`a, b`)` è quindi $\Theta(\log a \cdot (\log a + \log b)) = \Theta(\log^2 a + \log a \cdot \log b)$. Questa espressione assume valore massimo se a e b sono dello stesso ordine di grandezza, nel qual caso la complessità dell'algoritmo risulta:

$$T(n) = \Theta(n^2)$$

cioè quadratica nella *dimensione* (non nel *valore*) dei dati d'ingresso. Lasciamo al lettore il compito di verificare che l'usuale algoritmo di moltiplicazione ha la stessa complessità.

[5] Se si parte da 1 e si raddoppia il valore per n volte si ottiene 2^n. Poiché $\log_2 n$ è la funzione inversa di 2^n, se si dimezza un numero a per $\lfloor \log_2 a \rfloor$ volte si ottiene 1 (l'approssimazione inferiore è necessaria se a non è una potenza di 2).

Il precedente calcolo si è concluso con l'esame del *caso pessimo* in riferimento ai possibili valori assunti dai dati d'ingresso. È un approccio pessimistico, ma garantisce che l'algoritmo termini *sempre* entro il tempo calcolato. Per tener conto che la situzione potrebbe essere più favorevole possiamo esprimere la complessità di `Molt(`a, b`)` come $T(n) = O(n^2)$, che mostra come il valore n^2 non sia necessariamente sempre raggiunto. Questo spiega il senso dell'ordine O (o Ω) come limite superiore (o inferiore). Un'analisi alternativa molto più complessa si rivolge al *caso medio* e non sarà qui presa in considerazione perché non particolarmente rilevante ai nostri fini.

L'analisi di complessità dell'algoritmo `Euclid(`a, b`)` è piuttosto complicata e ci limitiamo a riportarne alcuni punti. La dimensione dell'ingresso è $n = \Theta(\log a + \log b) = \Theta(\log a)$ perché per ipotesi $a \geq b$. La complessità $T(n)$ è proporzionale al numero di chiamate ricorsive della funzione stessa, che dipende dal valore dei dati (se per esempio a è multiplo di b si ha a `mod` $b = 0$ e l'algoritmo termina alla seconda chiamata). Si può comunque dimostrare che in ogni chiamata risulta a `mod` $b < a/2$, quindi il parametro a si riduce almeno della metà ogni due chiamate: ne segue che il numero di chiamate è limitato da $O(\log a) = O(n)$. Ogni chiamata richiede di calcolare un modulo, e questa operazione può essere eseguita in tempo $\Theta(\log a \cdot \log b) = O(n^2)$ con l'usuale algoritmo di divisione (ma si può fare di meglio). In conclusione la complessità dell'algoritmo risulta:

$$T(n) = O(n^3)$$

cioè al più cubica nella dimensione dell'ingresso.

L'analisi degli algoritmi `Ricerca_Sequenziale` e `Ricerca_Binaria` è molto più semplice. Nella nostra ipotesi gli elementi dell'insieme su cui si esegue la ricerca sono memorizzati in un vettore A le cui celle hanno dimensione prefissata e indipendente dalla lunghezza del vettore stesso. Ai nostri fini la dimensione delle celle è quindi una costante, e la dimensione complessiva dell'ingresso è $n = \Theta(|A|)$: trascurando la costante moltiplicativa porremo direttamente $n = |A|$. L'algoritmo `Ricerca_Sequenziale` ripete al massimo $|A|$ volte un ciclo all'interno del quale esegue un confronto tra dati di lunghezza costante; quindi ogni iterazione richiede tempo costante e in complesso si ha:

$$T(n) = O(n).$$

Il limite superiore contenuto nella notazione O è giustificato dal fatto che l'algoritmo potrebbe arrestarsi in pochi passi se l'elemento cercato è in una delle prime posizioni del vettore, ma è necessaria l'intera scansione se tale elemento è l'ultimo o addirittura non è presente.

L'algoritmo `Ricerca_Binaria` è più interessante. Notiamo che esso contiene due chiamate ricorsive su $\lfloor n/2 \rfloor$ elementi in conseguenza del verificarsi di condizioni opposte: quindi in ogni caso si esegue solo una di tali chiamate. Ammettendo che il test

tra k e $A[m]$ abbia risultato ternario e quindi si esegua una volta sola per stabilire la relazione tra i due elementi, il numero totale $C(n)$ di confronti eseguiti è espresso dalla *equazione di ricorrenza*:

$$C(n) \begin{cases} = 1 & \text{se } n = 1, \\ \leq 1 + C(\lfloor n/2 \rfloor) & \text{se } n > 1. \end{cases}$$

Il valore limite $C(1) = 1$ indica che si esegue un solo confronto quando si raggiunge un sottoinsieme di un elemento, e il segno $\leq$ nell'equazione indica che non si esegue una successiva chiamata ricorsiva se si trova k. L'equazione si risolve facilmente sviluppando la serie:

$$C(n) \leq C(n/2) + 1 \leq C(n/4) + 1 + 1 \leq \ldots \leq C(1) + \ldots + 1 + 1 \leq 1 + \log_2 n$$

ove si è posto $C(1) = 1$, e il numero di uni nella somma è al più $\log_2 n$ come già osservato nell'analisi di `Molt`. In conclusione, ponendo che le singole operazioni contenute nell'algoritmo richiedano tempo costante, e che quindi la sua complessità $T(n)$ sia proporzionale al numero di confronti tra elementi, avremo:

$$T(n) = O(\log n).$$

Le notazioni Θ e O sono impiegate per esprimere la complessità di un algoritmo. La notazione Ω è invece necessaria per definire il *limite inferiore* alla *complessità di un problema*, cioè il tempo minimo necessario per *qualsiasi* algoritmo di soluzione. Per esempio la moltiplicazione tra due interi a, b si può eseguire in tempo $O(n^2)$ con l'algoritmo `Molt`: questo rappresenta un *limite superiore* alla complessità del problema nel senso che ci permette di escludere l'impiego di algoritmi di complessità maggiore. Poiché però per ottenere il prodotto si deve esaminare l'intera sequenza che rappresenta a e b, che ha lunghezza proporzionale a n, un limite inferiore per il problema è dato da $\Omega(n)$. In sostanza i limiti inferiori si riferiscono ai problemi, i limiti superiori agli algoritmi di soluzione.

Lo studio dei limiti inferiori è in genere difficile se si cercano risultati meno ingenui di quello appena citato. Ci limitiamo a un'osservazione semplice ma importante. Per un dato problema P, poniamo che L_s sia un limite superiore (relativo a un algoritmo A) e L_i sia un limite inferiore (relativo al problema stesso). Se $L_s = L_i$ l'algoritmo A si dice *ottimo*, ovviamente in ordine di grandezza: in linea di principio non è necessario cercare ulteriori algoritmi per P perché non sarebbero migliori di A. Per eseguire la moltiplicazione esistono algoritmi migliori di `Molt`, ma non esiste dimostrazione che alcuno di essi sia ottimo: il problema in sostanza è ancora aperto.

3.3 Algoritmi polinomiali e esponenziali

Nel paragrafo precedente abbiamo esaminato due algoritmi per la ricerca di un elemento in un insieme ottenendo, per il caso pessimo, tempi di $O(n)$ e $O(\log n)$. Tra queste funzioni vi è un divario che cresce grandemente all'aumentare di n. Ponendole a confronto, e assegnando per esempio la base 2 al logaritmo, abbiamo:

n:	1	2	4	8	16	...	1.024	...	1.048.576	...	1.073.741.824	...
$\log_2 n$:	0	1	2	3	4	...	10	...	20	...	30	...

In sostanza se il valore di n raddoppia il valore di $\log_2 n$ aumenta solo di uno. Questo dimostra che la ricerca binaria è estremamente più efficiente della ricerca sequenziale a meno che i valori di n siano molto piccoli, e spiega la comune esperienza che cercare un nome in un elenco telefonico di 100.000 o 200.000 utenti richiede praticamente lo stesso tempo.

Se si inverte la funzione logaritmo, cioè si passa da n a 2^n, si ottiene esattamente lo stesso effetto: la funzione esponenziale 2^n cresce incomparabilmente più in fretta della funzione lineare n, e crea problemi enormi se riferita alla complessità di un algoritmo. Nella terminologia corrente un algoritmo è detto *polinomiale* se la sua complessità $T(n)$ è di ordine $O(p(n))$, ove $p(n)$ è un polinomio arbitrario nella dimensione n dell'ingresso: tutti gli algoritmi esaminati fin qui sono polinomiali. Se un limite superiore polinomiale non esiste l'algoritmo è detto *esponenziale*, anche se la famiglia viene così a comprendere alcune funzioni "intermedie", come $n^{\log n}$, che in genere non si presentano nello studio della complessià. Tipiche funzioni esponenziali in cui potremo imbatterci sono $T(n) = \Theta(c^n)$ con $c > 1$ costante, $T(n) = \Theta(n^n)$, $T(n) = \Theta(n!)$; a proposito della terza ricordiamo l'esponenzialità della funzione fattoriale attraverso l'*approssimazione di Stirling*:

$$n! \simeq \sqrt{2\,\pi\,n}\,(n/e)^n \tag{3.2}$$

ove e è la base dei logaritmi naturali.

La distinzione, sopra ogni altra, tra le famiglie di algoritmi polinomiali e esponenziali è giustificata dalla pratica inconfrontabilità tra le leggi di crescita delle loro complessità. Per fare un grossolano esempio quantitativo poniamo che tre algoritmi A_1, A_2 e A_3 risolvano lo stesso problema su n dati con un numero di operazioni rispettivamente di n, n^2 e 2^n, e che il calcolatore esegua un miliardo di operazioni al secondo. Per $n = 50$ i calcoli sono portati a termine da A_1 in $5 \cdot 10^{-8}$ secondi, da A_2 in $2.5 \cdot 10^{-6}$ secondi, mentre A_3 richiede $\sim 10^{15}$ secondi ovvero trenta milioni di anni. L'algoritmo esponenziale è perciò utilizzabile solo per piccolissimi valori di n.

La differenza tra algoritmi polinomiali e esponenziali appare altrettanto drastica se si studia la dimensione dei dati praticamente trattabili in funzione dell'incremento di velocità dei calcolatori. Si potrebbe infatti pensare che quello che non è fattibile oggi possa esserlo in futuro, ma un semplice calcolo induce a credere che ciò non sia vero. Poniamo che un nuovo calcolatore esegua ogni operazione in un tempo pari a $1/k$ dell'attuale e che l'utente abbia a disposizione sempre lo stesso tempo di calcolo $\bar{t}$, e vediamo come varia la dimensione dei dati trattabili, che passerà da n_1 a n_2. Possiamo grossolanamente assumere che impiegare il nuovo calcolatore per un tempo $\bar{t}$ equivalga a impiegare il vecchio per un tempo $k\,\bar{t}$. Per un algoritmo polinomiale che risolve il problema su n dati in $c\,n^s$ secondi (con c ed s costanti) si ha:

$$c\,n_1^s = \bar{t},\ c\,n_2^s = k\,\bar{t} \ \Rightarrow\ n_2 = \sqrt[s]{k}\,n_1$$

con vantaggio tanto maggiore quanto più è basso il valore di s, cioè quanto migliore è l'algoritmo. Per esempio impiegando l'algoritmo quadratico A_2 discusso prima, un calcolatore mille volte più veloce consente di moltiplicare per $\sqrt{1000} \simeq 30$ il numero di dati trattabili a pari tempo di calcolo. Se però l'algoritmo scelto è esponenziale e risolve il problema in $c\,2^n$ secondi, si ha:

$$c\,2^{n_1} = \bar{t},\ c\,2^{n_2} = k\,\bar{t} \ \Rightarrow\ 2^{n_2} = k\,2^{n_1} \ \Rightarrow\ n_2 = n_1 + \log_2 k.$$

In questo caso il beneficio derivante dall'incremento di velocità è semplicemente *additivo* e cresce inoltre con estrema lentezza poiché ridotto dalla funzione logaritmo. Per l'algoritmo esponenziale A_3 discusso prima, un calcolatore un milione di volte più veloce consente solo di sommare $\log_2 1.000.000 \simeq 20$ al numero di dati trattabili a pari tempo di calcolo. In sostanza un algoritmo esponenziale in pratica *non consente* di trattare dati di dimensioni anche modeste, né sembra esservi speranza di cambiamento con i miglioramenti della tecnologia.[6] Dunque dovremmo evitare di costruire algoritmi esponenziali: ma questo, come vedremo, non sempre è possibile.

Nella maggioranza dei problemi che si affrontano con i calcolatori l'esponenzialità è legata all'impiego, a volte mascherato, di due strutture combinatorie di base: l'insieme delle 2^n configurazioni di n bit, e l'insieme delle $n!$ permutazioni di n elementi. Vediamo come tali strutture possano essere costruite algoritmicamente e quindi utilizzate. Dato un vettore A di lunghezza n, il seguente algoritmo `Configurazioni` costruisce in A le 2^n possibili configurazioni binarie (disposizioni con ripetizione dei due elementi 0 e 1 in gruppi di n). Per ogni configurazione l'algoritmo stabilisce se essa possiede determinate proprietà mediante una procedura `Controllo` che specificheremo nel seguito trattando di problemi concreti. Il calcolo è avviato con la chiamata iniziale `Configurazioni`$(A, 1)$ che innesca la costruzione di tutte le configurazioni binarie dalla $00\ldots0$ alla $11\ldots1$, facendo variare le cifre a partire dall'ultima.

[6] Similmente si dimostra che nessun aiuto può venire dal calcolo parallelo.

```
Procedure Configurazioni(A, k):
    for i ← 0 to 1 do
        A[k] ← i;
        if k = n then Controllo(A)
                 else Configurazioni(A, k + 1).
```

Se si vuole, la procedura `Configurazioni` genera tutti i numeri binari di n cifre e li "controlla" uno a uno per verificarne determinate proprietà. Questo suggerisce un immediato impiego dell'algoritmo in campo numerico. Consideriamo per esempio il *problema della primalità* $\mathcal{P}_{primo}(N)$ che come vedremo è essenziale in molte applicazioni crittografiche: dato un intero positivo N stabilire se è primo. Rappresentiamo N con n cifre binarie. Se N non è primo uno dei suoi divisori deve essere $\leq \sqrt{N}$, e si può quindi risolvere il problema provando se N è divisibile per tutti gli interi compresi tra 2 e $\sqrt{N}$, cioè rappresentati da sequenze binarie di lunghezza al massimo $n/2$. Si può allora impiegare l'algoritmo `Configurazioni` combinato con una procedura `Controllo` che, per ogni numero generato R, verifica se N è divisibile per R e in caso positivo arresta il calcolo con successo. Questo algoritmo ha complessità $O(2^{n/2} \cdot D(n))$, ove $D(n)$ è il costo (polinomiale) della divisione, ed è quindi esponenziale nella lunghezza della rappresentazione di Q. Poiché la crittografia opera su numeri binari di migliaia di cifre dovremo ricorrere ad altri metodi per stabilire la primalità.

Vediamo ora come la costruzione delle configurazioni binarie intervenga in problemi combinatori. Consideriamo il seguente *problema dello zaino* $\mathcal{P}_{zaino}(I, b, c)$, paradigma di riferimento per molti problemi di allocazione di risorse: dato un insieme I di n interi positivi (non necessariamente diversi tra loro), e altri due interi positivi b, c, stabilire se esiste un sottoinsieme di I la cui somma degli elementi sia compresa tra b e c. Intuitivamente c è il massimo peso che può sopportare lo zaino, I un insieme di oggetti di pesi vari, b il peso complessivo minimo che si vuole caricare.[7] Poniamo che gli elementi di I siano memorizzati in un vettore E e introduciamo un vettore binario A *di appartenenza*, anch'esso di n elementi, per descrivere arbitrari sottoinsiemi $T \subseteq I$: per $1 \leq i \leq n$, $A[i] = 1$ indica che $E[i] \in T$, $A[i] = 0$ indica che $E[i] \notin T$. Poniamo per esempio:

E:	5	8	12	6	6	7	25	4	2	9
A:	0	1	0	1	0	1	0	0	1	0

[7] Questa versione del problema è un po' semplificata rispetto a quella classica, ma è perfettamente adatta ai nostri scopi.

Il vettore E corrisponde a un'istanza del problema dello zaino per $n = 10$ elementi, e il vettore A corrisponde al sottoinsieme di elementi $T = \{8, 6, 7, 2\}$. Ponendo $b = 20$ e $c = 24$ la configurazione binaria contenuta in A specifica una soluzione, poiché la somma degli elementi di T è pari a 23.

Poiché le configurazioni di A corrispondono ai sottoinsiemi di I (che sono infatti 2^n), esse possono essere costruite con l'algoritmo `Configurazioni`. Per risolvere il problema $\mathcal{P}_{zaino}(I, b, c)$ è allora sufficiente specificare la procedura di controllo come:

`Procedure Controllo(`A`)`:
if $b \leq \sum_{i=1}^{n} A[i]E[i] \leq c$ **then success**.

ove il comando **success** arresta il calcolo decretando che esiste una soluzione (se il comando non viene incontrato, sono state provate senza successo tutte le configurazioni e la soluzione non esiste). La complessità dell'algoritmo così costruito è $T(n) = O(2^n \cdot S(n))$, ove $S(n)$ è il costo (polinomiale) della sommatoria, ed è quindi esponenziale nel numero di elementi di I, cioè nella dimensione dell'ingresso.

Il problema $\mathcal{P}_{zaino}(I, b, c)$ è stato posto in forma decisionale perché, come vedremo, questa forma è sufficiente a decretarne il grado di difficoltà. In pratica si può richiedere di comunicare all'esterno la soluzione trovata, o la soluzione *ottima* (cioè il sottoinsieme di somma massima contenuto nello zaino): si tratta di versioni standard del problema, che si incontrano per esempio nell'allocazione di file in un disco. Lo schema di calcolo è sempre lo stesso e la complessità è ovviamente sempre esponenziale; per esempio la soluzione ottima si determina ponendo inizialmente $max \leftarrow 0$, quindi chiamando la procedura `Configurazioni` e specificando la procedura di controllo come:

`Procedure Controllo(`A`)`:
if $(b \leq \sum_{i=1}^{n} A[i]E[i] \leq c)$ **and** $(max < \sum_{i=1}^{n} A[i]E[i])$
then $max \leftarrow \sum_{i=1}^{n} A[i]E[i]$;
for $i \leftarrow 1$ **to** n **do** $\bar{A}[i] \leftarrow A[i]$.

Il connettivo **and** tra due condizioni indica che entrambe devono essere verificate per eseguire la parte **then**. Il valore finale della variabile max è pari alla somma degli elementi nella soluzione ottima, e questa è registrata nel vettore di servizio $\bar{A}$. Se $max = 0$ non esiste alcuna soluzione.

La soluzione proposta per il problema dello zaino può apparire irragionevole in quanto non sfrutta alcuna proprietà del problema ma procede per enumerazione completa dei casi. Ed è proprio da questa enumerazione che discende l'esponenzialità dell'algoritmo. Alcune considerazioni locali possono far diminuire il numero di casi da esaminare: per esempio nell'istanza riportata sopra l'elemento 25 potrebbe es-

sere eliminato prima di procedere perché è maggiore di $c = 24$. Ma nella grande maggioranza delle situazioni, e certamente in quella più sfavorevole, il numero di configurazioni possibili rimane esponenziale in n e non si conosce strategia migliore che esaminarle tutte. In sostanza, come vedremo nel prossimo paragrafo, il problema stesso sembra avere complessità esponenziale anche se non è mai stato dimostrato un tale limite inferiore.

Un'altro problema combinatorio di base è quello della costruzione di permutazioni. Dato un insieme di n elementi contenuti in un vettore P, l'algoritmo seguente costruisce in P tutte le $n!$ permutazioni di tali elementi. Come nel caso precedente, per ogni permutazione l'algoritmo stabilisce se essa possiede determinate proprietà mediante una procedura `Controllo`.

```
Procedure Permutazioni(P, k):
    if k = n then Controllo(P)
            else for i ← k to n do
                    scambia P[k] ↔ P[i];
                    Permutazioni(P, k + 1);
                    scambia P[k] ↔ P[i].
```

Il calcolo è avviato con la chiamata iniziale `Permutazioni`$(P, 1)$. L'algoritmo è basato sull'osservazione che le permutazioni di n elementi possono essere divise in gruppi, ponendo in ciascun gruppo quelle che iniziano con il primo, il secondo,..., l'n-esimo elemento ciascuno seguito dalle permutazioni degli altri $n - 1$. Nel primo gruppo troviamo $P[1]$ seguito da tutte le permutazioni di $P[2], \ldots, P[n]$; nel secondo troviamo $P[2]$ seguito da tutte le permutazioni di $P[1], P[3], \ldots, P[n]$ e così via. Questa definizione è induttiva e la correttezza dell'algoritmo si può quindi dimostrare per induzione: occorre solo notare che la seconda operazione di scambio tra $P[k]$ e $P[i]$ è necessaria per ripristinare l'ordine degli elementi dopo la costruzione ricorsiva delle permutazioni degli elementi che seguono il primo. Più difficile è capire il funzionamento dell'algoritmo e individuare l'ordine in cui vengono costruite le permutazioni. Invitiamo il lettore a rifletterci un momento simulando a mano il comportamento dell'algoritmo sull'insieme $\{1, 2, 3\}$: otterrà nell'ordine le permutazioni: 1,2,3 - 1,3,2 - 2,1,3 - 2,3,1 - 3,2,1 - 3,1,2, e con l'ultima operazione di scambio ripristinerà l'ordine iniziale 1,2,3 degli elementi.

La costruzione di permutazioni interviene per esempio nel *problema del ciclo hamiltoniano* $\mathcal{P}_{ham}(G)$, paradigma di riferimento per molti problemi di percorsi. Ricordiamo anzitutto che un grafo $G = (V, E)$ è composto da un insieme V di vertici, e un insieme E di spigoli che connettono coppie di vertici. La struttura può essere convenientemente descritta rappresentando i vertici con gli interi $1, \ldots, n$, con $n = |V|$, e

rappresentando E con una matrice binaria A *di adiacenza*, di dimensioni $n \times n$, ove si pone $A[i,j] = 1$ se esiste uno spigolo che connette il veritice i al vertice j, $A[i,j] = 0$ se tale spigolo non esiste. La dimensione della descrizione di G è dunque di ordine $\Theta(n \log n)$ per rappresentare V più $\Theta(n^2)$ per rappresentare E: quindi in totale è di ordine $\Theta(n^2)$. Un *percorso* in un grafo è una sequenza di vertici $v_1, v_2, \ldots, v_k$ tale che esiste lo spigolo da v_i a v_{i+1} per ogni coppia di vertici consecutivi nel percorso. Se v_k coincide con v_1 il percorso è detto *ciclo*. Dato un grafo arbitrario G il problema $\mathcal{P}_{ham}(G)$ chiede di stabilire se esiste un ciclo in G che contiene tutti i vertici esattamente una volta.

Se associamo al grafo G un vettore P contenente gli interi tra 1 e n, una permutazione di P rappresenta una sequenza che contiene tutti i vertici: la permutazione rappresenta quindi una soluzione di $\mathcal{P}_{ham}(G)$ (cioè un ciclo hamiltoniano) se nella matrice di adiacenza di A si ha $A[P[i], P[i+1]] = 1$ per ogni $1 \leq i \leq n-1$, e $A[P[n], P[1]] = 1$. Si possono allora costruire tutte le permutazioni in P con l'algoritmo `Permutazioni` e risolvere $\mathcal{P}_{ham}(G)$ specificando la procedura di controllo come:

`Procedure Controllo(`P`)`:
if $(A[P[i], P[i+1]] = 1,\ 1 \leq i \leq n-1)$ **and** $(A[P[n], P[1]] = 1)$
then success.

Come visto in precedenza il comando **success** arresta il calcolo decretando che esiste una soluzione. La complessità dell'algoritmo così costruito è $T(n) = O(n \cdot n!)$ ove il fattore n nel prodotto rappresenta il tempo per il controllo nella matrice A, ed è quindi esponenziale nella dimensione dell'ingresso.

Il problema $\mathcal{P}_{ham}(G)$ è stato posto in forma decisionale ma in pratica si può richiedere di comunicare all'esterno la soluzione trovata, o una soluzione con determinate proprietà. In una sua famosa versione che prende il nome di *problema del commesso viaggiatore* $\mathcal{P}_{cv}(G)$, si assegna una *lunghezza* a ogni spigolo del grafo e si chiede di trovare il ciclo hamiltoniano (se ne esiste uno) di minima lunghezza complessiva: si tratta per esempio di stabilire il percorso più breve per la testa di un trapano a controllo numerico con cui si devono fare molti fori in un pezzo meccanico. Lo schema di calcolo è sempre lo stesso e la complessità è ovviamente sempre esponenziale: anche $\mathcal{P}_{ham}(G)$ non sembra risolubile in tempo polinomiale, ma non si conosce dimostrazione in proposito.

3.4 Le classi P, NP, co-NP e NPC

Dopo aver visto come risolvere alcuni problemi in tempo polinomiale e aver proposto per altri solo algoritmi esponenziali, dobbiamo fare un po' di ordine nella teoria.