INTRODUZIONE ALLA

COMPLESSITÀ DI CALCOLO

Dispense del corso di Fondamenti di Informatica II

Fabrizio d'Amore, Alberto Marchetti Spaccamela

A.A. 2018-2019

Indice

Prefazione

Una domanda fondamentale che coinvolge tutte le persone interessate alla programmazione è conoscere le potenzialità e i limiti degli elaboratori. In particolare vogliamo sapere quali sono i problemi che si possono effettivamente risolvere eseguendo un programma ad un elaboratore rispondendo alla seguente domanda:

COSA PUÒ ESSERE CALCOLATO DA UN CALCOLATORE E QUANTO COSTA FARLO?

La prima parte della domanda (Cosa può essere calcolato da un calcolatore) fu studiata ben prima che fosse disponibile un calcolatore reale (almeno come viene inteso ai giorni nostri): negli anni trenta del secolo scorso diversi ricercatori si chiesero quale fosse il vero significato della parola *"calcolabile"*. Il loro interesse non era limitato ai calcoli matematici ma a quali problemi potessero essere risolti nell'ambito della matematica e della logica. Questi ricercatori hanno formulato i risultati principali della *teoria della calcolabilità*; in particolare hanno proposto e studiato le potenzialità di modelli di calcolo, ciascuno specificato da un insieme di operazioni (o istruzioni) ammissibili e da regole con cui scrivere ed eseguire i programmi.

I ricercatori interessati alla calcolabilità avevano due obiettivi principali:

1. stabilire i limiti dei modelli di calcolo e stabilire se esistevano calcoli "impossibili" per i diversi modelli di calcolo;

2. individuare gli elementi essenziali del calcolo, cioè erano interessati a definire modelli di calcolo molto semplici che tuttavia avessero la capacità di risolvere tutti i problemi risolubili con modelli più complessi.

Per quanto riguarda il primo punto, sulla scia di uno dei più famosi risultati della logica matematica (il teorema di Gödel) è stato possibile stabilire che i modelli di calcolo e i metodi algoritmici non sono in grado di risolvere tutti i problemie che quindi

ESISTONO PROBLEMI CHE NON POSSONO ESSERE RISOLTI DA UN ELABORATORE

Per quanto riguarda il secondo punto sono stati proposti e analizzati diversi modelli di calcolo; l'utilizzo di un dato modello di calcolo M per la risoluzione di un problema matematico comportava la scrittur di un programma scritto nel linguaggio del modello considerato che specifica la sequenza di operazioni da eseguire per risolvere il problema. All'epoca le operazioni specificate dal modello di calcolo non erano fisicamente realizzabili da una macchina come un calcolatore ma i suoi calcoli potevano essere eseguiti manualmente; i modelli di calcolo proposti da questi studiosi sono detti modelli di calcolo astratti perché non richiedono l'esistenza di macchine che eseguono i programmi.

La teoria della calcolabilità è interessata a caratterizzare cosa possiamo calcolare ma non considera quanto sia costoso (in termini di tempo e di memoria) risolvere i problemi. In questa dispensa ci occupiamo della *teoria della complessità* che studia i problemi che possiamo risolvere in modo efficiente cercando di capire quanto costa l'esecuzione di un programma e di caratterizzare i problemi per cui esiste un programma efficiente e quelli per cui tale programma non esiste. Siamo cioè interessati a sapere cosa possiamo effettivamente calcolare in pratica; infatti, non ha molto senso affermare che un problema è risolubile se la sua soluzione richiede di utilizzare tutti i calcolatori del pianeta per mille anni. La teoria si occupa anche di definire i possibili approcci nei casi di problemi che sono difficili.

Nel capitolo 1 di queste dispense vedremo

1) COME DEFINIRE IN TERMINI MATEMATICI IL COSTO DI UN ALGORITMO OTTENENDO UNA VALUTAZIONE INDIPENDENTE DAL LINGUAGGIO DI PROGRAMMAZIONE USATO PER LA SUA IMPLEMENTAZIONE, DALL'ELABORATORE CHE UTILIZZIAMO E DAI DATI DI PROVA

Nel capitolo 2 vedremo che

2) NON TUTTI I PROBLEMI SONO RISOLUBILI IN MODO EFFICIENTE: MOLTI PROBLEMI NON POSSONO ESSERE RISOLTI IN MODO EFFICIENTE.

La trattazione che faremo nel seguito per rispondere alla domanda iniziale è limitata e non entra nei dettagli di una teoria che non ha solo interesse teorico ma che per sua natura cerca di migliorare lefficienza dei programmi e di trovare soluzioni per affrontare problemi computazionalmente complessi. Non sorprendentemente esistono numerosi libri di testo dedicati agli argomenti trattati in queste dispense. in particolare, i seguenti volumi hanno ispirato in parte la stesura delle dispense stesse e sono consigliati come letture aggiuntive:

- P. Crescenzi, *Informatica teorica*, dispense disponibili sul sito dell'autore (U. Firenze).
- F. d'Amore....
- D. Harel, Y.Feldman, *Algoritmi, Lo spirito dell'informatica*, Springer 2007.

Capitolo 1

Analisi della complessità di calcolo

1.1 Introduzione

L'analisi della complessità è anche uno strumento che ci consente di spiegare come si comporta un algoritmo man mano che l'input aumenta. Se forniamo un input diverso, come si comporta l'algoritmo? Se il nostro algoritmo impiega 1 secondo per eseguire un input di dimensione 1000, come si comporterà se raddoppierò la dimensione di input? Funzionerà altrettanto veloce, metà veloce o quattro volte più lento? Nella programmazione pratica, questo è importante in quanto ci consente di prevedere come si comporterà il nostro algoritmo quando i dati di input diventeranno più grandi. Ad esempio, se abbiamo realizzato un algoritmo per un'applicazione web che funziona bene con 1000 utenti e ne misura il tempo di esecuzione, utilizzando l'analisi della complessità dell'algoritmo, possiamo avere una buona idea di cosa accadrà quando otterremo invece 2000 utenti. Per le competizioni algoritmiche, l'analisi della complessità ci fornisce informazioni su quanto a lungo il nostro codice verrà eseguito per i test più grandi utilizzati per verificare la correttezza del nostro programma. Quindi, se abbiamo misurato il comportamento del nostro programma per un piccolo input, possiamo avere una buona idea di come si comporterà per input più grandi. Iniziamo con un semplice esempio: trovare l'elemento massimo in una matrice.

In informatica la complessità di calcolo, (o complessità computazionale, o anche semplicemente la complessità) di un algoritmo è la quantità di risorse richieste per la sua esecuzione. La complessità computazionale di un problema è la minima complessità di un algoritmo (noto o ignoto) che risolve il problema. In questo modo si stabilisce una misura di *efficienza*,[1] valutando la capacità di un algoritmo di sfruttare al meglio le risorse impiegate. Le risorse di calcolo a cui siamo maggiormente interessati sono il tempo di calcolo (o semplicemente tempo) e la quantità di memoria utilizzata (spazio)

Lo scopo dell'analisi della complessità di calcolo è studiare la complessità di problemi ed algoritmi con l'obiettivo principale di trovare algoritmi di soluzione sempre più efficienti. Intuitivamente, per algoritmo efficiente intendiamo un programma che richiede una quantità limitata di risorse di calcolo per la sua esecuzione. Le risorse di calcolo che si considerano sono il *tempo di elaborazione* richiesto e la *quantità di memoria* necessaria.

Delle due risorse *tempo* e *spazio*, la più importante, che considereremo nel seguito, è il tempo di calcolo richiesto. Le ragioni sono che nella pratica la risorsa tempo è quasi sempre la risorsa critica; la continua riduzione dei costi delle memorie sia volatili che permanenti è una delle ragioni per cui siamo interessati maggiormente al tempo di calcolo[2].

[1] Non si confondano le *qualità* degli algoritmi con le loro *proprietà*: mentre queste ultime debbono essere sempre soddisfatte (ad es., la correttezza, la finitezza ecc.), le prime possono anche non esserlo, ma aumentano senza dubbio il valore degli stessi, anche in termini di produttività e ottimizzazione delle risorse.

[2] Ovviamente ci sono casi in cui l'efficienza in termini di spazio è fondamentale. Ad esempio, nel caso degli smartphone, ridurre i tempi di calcolo significa minor dispendio energetico, limitare le necessità in termini di spazio può incrementare il numero effettivo di dispositivi capaci di eseguire l'applicazione. Considerazioni analoghe valgono per i sistemi embedded utilizzati nell'Internet delle cose (Internet of Things - IoT) in cui si utlizzano processori di

Si noti inoltre che, poiché gli algoritmi sono programmi che eseguono solo un calcolo, e non altre cose che i computer spesso fanno come attività di rete o input e output dell'utente, l'analisi della complessità ci consente di misurare la velocità di un programma quando esegue calcoli. Esempi di operazioni che non sono puramente computazionali sono le operazioni di accesso alla rete come la trasmissione e la ricezione di dati attraverso internet.

Lo studio della complessità di calcolo non ha solo interesse teorico. Infatti, in molti casi le considerazioni sull'efficienza temporale e spaziale degli algoritmi guidano la scelta del progettista alla soluzione software, assieme ovviamente a vincoli e requisiti provenienti dal dominio applicativo. L'obiettivo è dunque abilitare il progettista a disegnare la soluzione software ottimizzando l'impiego delle risorse e lo strumento per raggiungere questo scopo è la capacità di confrontare algoritmi.

Nel seguito di questo capitolo vedremo i concetti fondamentali che costituiscono le basi per analizzare la complessità degli algoritmi. Applicheremo questi concetti analizzando e confrontando diversi algoritmi di soluzione per lo stesso problema. Infine concentreremo la nostra attenzione sulla complessità computazionale dei problemi mostrando come sia possibile in alcuni casi stabilire come possa ...

1.2 Valutazione dell'efficienza degli algoritmi

Intuitivamente, un programma è più *efficiente* di un altro se la sua esecuzione richiede meno risorse di calcolo. Come abbiamo accennato considereremo nel seguito solamente il *t*empo di calcolo, che rappresenterà l'unica grandezza che ci interessa ed in base alla quale valuteremo *l'efficienza* (o la *complessità*) di calcolo del programma.

Se potessimo misurare il tempo di calcolo in unità standard come i secondi, allora la valutazione del costo di un algoritmo sarebbe semplice: basterebbe eseguire un programma che realizza l'algoritmo disponendo di un orologio con cui misurare il tempo necessario per l'esecuzione[3]. Il tempo impiegato ci darebbe una misura della complessità dell'algoritmo; inoltre, avendo due algoritmi che risolvono lo stesso problema, potremmo stabilire qual è il più efficiente semplicemente eseguendoli i e confrontando i rispettivi tempi di esecuzione.

Purtroppo questo metodo di valutazione non è attendibile se non consideriamo le condizioni in cui si effettuano le prove. Infatti, bisogna tener conto:

- dell'*elaboratore* su cui il programma viene eseguito; in particolare il confronto di due programmi è valido solo se li eseguiamo con lo stesso elaboratore;
- del linguaggio di programmazione. Infatti è abbastanza probabile che, quando i dati di ingresso non sono grandi, un cattivo algoritmo scritto in un linguaggio di programmazione di basso livello come l'assembly sia più veloce di un buon algoritmo scritto in un linguaggio di programmazione di alto livello come Python. Inoltre se usiamo lo stesso linguaggio dobbiamo tenere conto del particolare compilatore usato per la traduzione del programma. Infatti compilatori diversi generano programmi in linguaggio macchina con caratteristiche di efficienza diverse; pertanto il confronto è valido solo se i programmi sono tradotti dallo stesso compilatore o interprete;
- dei *dati di ingresso* ai due programmi; per trarre una conclusione valida è necessario che i due programmi ricevano in ingresso gli stessi dati;
- della *significatività dei dati di ingresso* poiché, per stabilire quale dei due programmi sia più efficiente non è sufficiente eseguire i due programmi una sola volta, ma più volte con dati differenti; ma dopo quante volte possiamo terminare il confronto e decidere quale sia il programma più efficiente?

costo ridotto di produzione (anche inferiore ad un euro) ma con limitate risorse di memoria

[3]Esistono programmi chiamati profiler che misurano il tempo di esecuzione di un proogramma in millisecondi e possono aiutarci a ottimizzare il nostro codice individuando i colli di bottiglia.

La discussione precedente contrasta con l'obiettivo di formalizzare l'efficienza di un algoritmo come sua qualità intrinseca. In particolare, se non si tengono in conto le osservazioni precedenti può accadere che persone diverse giungano a conclusioni diverse sull'efficienza di due algoritmi; può accadere, infatti, che le prove effettuate da una persona mostrino che il primo programma sia più efficiente, mentre una seconda persona può giungere a conclusioni opposte sulla base delle sue prove. Pertanto concludiamo che *utilizzando unità di tempo standard come i secondi non otteniamo una valutazione oggettiva del costo di esecuzione di un programma.*

Vogliamo che la complessità dell'algoritmo permetta di confrontare due algoritmi a livello di idea - ignorando dettagli di basso livello come il linguaggio di programmazione dell'implementazione, l'hardware su cui si basa l'algoritmo o il set di istruzioni della CPU fornita. Vogliamo confrontare gli algoritmi in termini di ciò che sono: idee su come viene calcolato qualcosa.

L'analisi della complessità è anche uno strumento che ci consente di spiegare come si comporta un algoritmo man mano che l'input aumenta. Se forniamo un input diverso, come si comporta l'algoritmo? Se il nostro algoritmo impiega 1 secondo per eseguire un input di dimensione 1000, come si comporterà se raddoppierò la dimensione di input? Impiegherà lo stesso tempo, la metà o sarà quattro volte più lento?

Nella pratica, saper rispondere a questa domanda è importante in quanto ci consente di prevedere come si comporterà il nostro algoritmo quando i dati di input diventeranno più grandi. Supponiamo ad esempio di aver programmato un algoritmo per un'applicazione web per cui facciamo test di prova per verificare la correttezza del nostro programma con 100 utenti. Utilizzando l'analisi della complessità dell'algoritmo, possiamo avere una buona idea di cosa accadrà quando avremo invece 1000 o 10000 utenti senza necessariamente eseguire dati di test di queste dimensioni.

1.2.1 Calcolo del costo di esecuzione di un programma

Vedremo nel seguito un metodo di analisi che permette di esprimere valutazioni oggettive che non dipendono dal particolare elaboratore a disposizione, dal compilatore usato o dai dati di input utilizzati. In particolare vedremo come sia possibile esprimere il costo di un algoritmo tramite una funzione matematica il cui argomento è dato dalle dimensioni dell'input del problema.

Per ottenere una misura del costo che non dipende dal particolare elaboratore utilizzato valutiamo il numero di operazioni elementari effettuate da una macchina ispirata all'architettura di von Neumann e completamente dedicata: in tal modo ci si libera dalla necessità di fissare caratteristiche hardware/software di riferimento, che esegue il programma. [4]

Avendo in mente l'architettura di von Neumann - si veda lo schema in Fig. 1.1), si introduce dunque una macchina astratta, qui denominata CEM (C Elementary Machine), rispetto cui svolgere le analisi di costo temporale.

La differenza fondamentale fra la CEM e la macchina di von Neumann consiste nella capacità di eseguire algoritmi descritti in un linguaggio ad alto livello. In quanto segue si farà riferimento al linguaggio C, ma la trattazione può essere facilmente adattata a qualunque altro linguaggio ad alto livello o anche al cosiddetto pseudo-codice.

Definizione 1. Per la valutazione del costo di un algoritmo si assume che ogni operazione elementare abbia costo unitario. Il costo di una istruzione composta è pari alla somma dei tempi relativi alle sue singole componenti. Per le istruzioni di controllo di flusso e le istruzioni di ciclo i costi da imputare sono unitari per il test e tengono conto del numero di istruzioni elementari eseguite.

In particolare

[4] Per la descrizione originale dell'architettura si può consultare [**?**], dove viene ripreso il testo originale di John von Neumann, opportunamente riveduto e corretto.

[5] Immagine tratta da Wikipedia (licenza CC BY-SA 3.0 tramite Wikimedia Commons - `https://commons.wikimedia.org/wiki/File:Von_Neumann.gif#/media/File:Von_Neumann.gif`).

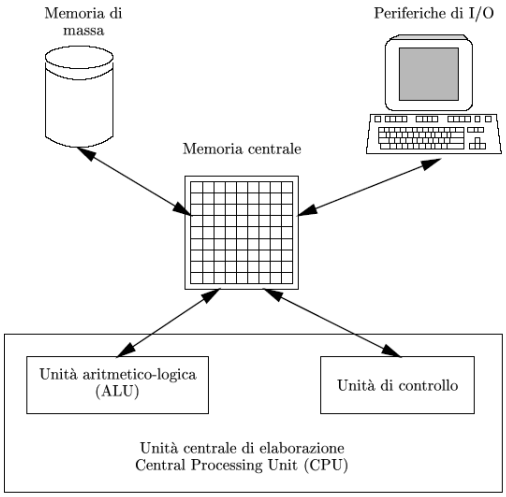

Figura 1.1 Architettura classica di Von Neumann.[5]

- il costo di esecuzione di ogni *istruzione semplice* (istruzione di assegnazione, di lettura, di scrittura) è pari al numero di celle di memoria a cui l'istruzione assegna un valore. Tutte le istruzioni semplici della nostra macchina hanno costo unitario. [6];
- il costo di un'istruzione di *ciclo* è dato dalla somma del costo totale di esecuzione del test di fine ciclo e dal costo totale di esecuzione del corpo del ciclo. Assumiamo che il costo di esecuzione del test sia unitario e, quindi, il primo termine è pari al numero di volte che il ciclo viene ripetuto. Il secondo termine è pari alla somma dei costi di esecuzione delle istruzioni del corpo del ciclo, tenendo conto di quante volte ciascuna istruzione del ciclo è eseguita.
- il costo di un'*istruzione condizionale* di tipo `if` è dato dal costo di esecuzione del test (calcolato con le medesime regole del caso precedente) più il costo di esecuzione della istruzione che viene eseguita se la condizione è vera. In modo analogo si valuta il costo di esecuzione di un'istruzione `if else`; in questo caso abbiamo il costo di verifica del test e poi il costo delle istruzioni eseguite se la condizione è vera oppure il costo delle istruzioni se la condizione è falsa.
- il costo di esecuzione di una *attivazione di sottoprogramma (o procedura o funzione)* è pari al costo di esecuzione di tutte le istruzioni che compongono il sottoprogramma, tenendo eventualmente conto del fatto che all'interno del sottoprogramma possono essere presenti altre chiamate di sottoprogramma.

Tab. 1.1 illustra una guida non completa che illustra i costi da associare alle varie tipologie di istruzione.

Se da un lato può apparire convincente l'idea di misurare i tempi di esecuzione degli algoritmi contando le istruzioni elementari eseguite da una macchina di von Neumann, dall'altro appare certamente sorprendente il modello di conteggio proposto con la CEM poiché contenente alcune assunzioni e semplificazioni che sembrano arbitrarie, se non addirittura grossolane.

Ad esempio, con una CEM, affermiamo che hanno lo stesso costo (unitario) le istruzioni

```
x = 1
```

che richiede una assegnazione, e

```
a[j-1] = x * j + (beta/ 2) - gamma /(3 * delta +8)
```

[6] Osserviamo che possiamo avere istruzioni di assegnazione che hanno costi maggiori di uno. Ad esempio, con riferimento al linguaggio Python, il costo di istruzione $x = 1$ è pari a uno, mentre il costo dell'istruzione $x = (1, 2, 3, 4)$ che aggiorna il valore di quattro celle di memoria è 4

Tabella 1.1. Sommario costi relativi alle istruzioni C.

costrutto C	costo
valutazione di espressione semplice (che non invoca funzioni)	unitario
valutazione di espressione complessa (che invoca funzioni)	somma costi funzioni chiamate
assegnazione a variabile o a elemento di array/struct	[costo espressione assegnata] + [costo eventuale espressione che definisce indice di array (trascurabile se unitario)]
istruzione composta	somma costi istruzioni componenti
chiamata di funzione	costo della funzione (unitario per `malloc`/`free`; dimensione dell'area allocata per `calloc`)
`return <expr>;`	[costo valutazione `<expr>`]
`if(<expr>)...`	[costo valutazione `<expr>`] + [costo istruzione eseguita]
`switch(<expr>)...`	[costo valutazione `<expr>`] + [somma costi istruzioni eseguite]
`while(<expr>)...` (con p iterazioni)	[$(p+1)\cdot$ (costo valutazione `<expr>`)] + [$p\cdot$ (costo istruzione eseguita)]
`do...while(<expr>);` (con p iterazioni)	[$p\cdot$ (costo valutazione `<expr>`)] + [$p\cdot$ (costo istruzione eseguita)]
`for(<ist1>;<expr>; <ist2>)...` (con p iterazioni)	[costo `<ist1>`] + [$(p+1)\cdot$ (costo valutazione `<expr>`)] + [$p\cdot$ (costo `<ist2>`)] + [$p\cdot$ (costo istruzione eseguita)]

che richiede un numero maggiore di operazioni tra cui moltiplicazioni e divisioni che richiedono un maggior tempo di calcolo della assegnazione.

Pertanto la nostra affermazione che le due istruzioni hanno lo stesso costo può apparire addirittura errata, poiché il numero di operazioni elementari associate alle due istruzioni sarà evidentemente significativamente differente.

Osserviamo però che, se indichiamo con t_1 il tempo di esecuzione della prima istruzione e con t_2 quello della seconda istruzione, esiste una costante c tale che

$$c \times t_1 > t_2$$

In altre parole t_1 è maggiore di t_2 al più per un fattore moltiplicativo costante c. Quindi la nostra ipotesi di assumere il coste delle due istruzioni uguale è esatta a meno di un fattore costante. Un ragionamento analogo può essere fatto per una qualunque altra istruzione, ottenendo, in generale, costanti diverse.

*U*n importante vantaggio di queste semplificazioni è quello di poter prescindere dal linguaggio di programmazione in cui viene scritto il programma e di poter far riferimento direttamente all'algoritmo descritto in linguaggio naturale.

Ovviamente in questo caso dobbiamo essere accorti e non specificare funzioni il cui costo di esecuzione una volta tradotte in un linguaggio sia unitario. Ad esempio, in Python il costo di esecuzione delle operazioni su stringhe *non* può essere considerato in generale pari a uno.

1.2.2 Analisi nel caso peggiore in funzione delle dimensioni dell'input

Dopo aver visto come si calcola la funzione che esprime il costo di un algoritmo o di un programma per un particolare ingresso, vediamo ora come definire una funzione che esprime il costo di un programma per ogni possibile ingresso.

Osserviamo che il costo di esecuzione di un programma dipende quasi sempre dai particolari dati di ingresso. Ad esempio, il costo di esecuzione di un programma di ordinamento di un insieme di numeri dipende dalle dimensioni dell'insieme che si considera: è ragionevole assumere che il programma impieghi meno tempo ad ordinare un insieme di 10 elementi che un insieme

di 10000 elementi. Pertanto, per tener conto del numero di dati con cui si esegue il programma assumiamo che *la dimensione dell'input rappresenta l'argomento della funzione che esprime il costo di esecuzione di un programma.*

Per *dimensione (o taglia) dell'input* si intende la quantità di memoria necessaria per memorizzare i dati di input del problema. Ad esempio, per il problema dell'ordinamento di un insieme, il numero di elementi dell'insieme da ordinare rappresenta la dimensione dell'input.

Si noti che in molti casi il costo di esecuzione del programma dipende non solo dalle dimensioni dell'input, ma anche dai particolari dati di ingresso. Generalmente possiamo distinguere diversi casi: il *caso migliore* (quello meno costoso), il *caso peggiore* (quello più costoso), il caso medio.

Per ottenere valutazioni oggettive sul costo di esecuzione di un algoritmo, la valutazione che si fornisce non deve dipendere dai particolari valori dei dati di ingresso o da particolari casi fortunati. In particolare faremo la seguente ipotesi che fa riferimento al caso peggiore.

Definizione 2. Valutazione del costo di un programma. Il costo di un programma viene valutato in funzione delle dimensioni dell'input con riferimento al **caso peggiore**, cioè quello in cui l'esecuzione impiega più tempo e assumendo che il costo di ogni singola istruzione sia pari a uno.

Siamo a questo punto in grado di fornire una valutazione nel caso di un semplice esempio di algoritmo di ricerca esaustiva.

Esempio 1. Supponiamo di dover effettuare la ricerca di un elemento in un insieme di elementi. Un possibile modo di rappresentazione dell'insieme utilizza un vettore a e memorizza ciascun elemento dell'insieme in una componente del vettore. Gli elementi dell'insieme sono memorizzati in un ordine qualunque. In questo caso la dimensione dell'input è data dal numero di elementi del vettore.

La figura seguente fornisce un'implementazione di un semplice algoritmo che scansiona le componenti del vettore fino a quando non individua il valore cercato o quando tutti gli elementi del vettore sonos tati considerati.

```
 1. int ricercaSequenziale(int a[], int n, int k) {
 2. /*
 3.   restituisce indice della posizione di k
 4.   oppure -1 se k non presente; assume n > 0
 5. */
 6.   int i = 0;
 7.   while((i < n) && (a[i] < k)) i = i + 1;
 8.   if((i == n) || (a[i] > k) return -1;
 9.   else return i;
10. }
```

Nel caso della ricerca esaustiva dell'esempio precedente il costo di esecuzione dipende evidentemente dal particolare elemento che si cerca: se l'elemento cercato è il primo elemento del vettore, allora si effettua un solo confronto; se l'elemento cercato è il secondo elemento del vettore, allora si effettuano due confronti; analogamente se l'elemento cercato è l'i-esimo elemento del vettore, allora si effettuano i confronti. Pertanto, il caso peggiore è quello della ricerca di un elemento che si trova nell'ultima posizione del vettore, oppure di un elemento non presente nel vettore, perché l'algoritmo deve esaminare tutte le componenti del vettore.

In particolare abbiamo la seguente valutazione di costo nel caso peggiore:

- le assegnazioni nelle linee `6.`, `16.` e `18.` hanno tutte costo unitario;
- il ciclo `while` a linea `7.` ha costo pari a $2p + 1$, essendo p il numero di iterazioni eseguite, $0 \leq p \leq \mathtt{n}$; quindi il costo massimo sarà $2\mathtt{n} + 1$;

- l'istruzione condizionale nelle linee `8.` e `9.` ha costo pari a 2 (l'espressione valutata è semplice e ha costo unitario; il `return` eseguito ha costo unitario);

Sulla base di questa discussione si può dunque concludere che su una CEM il costo temporale nel caso peggiore di `ricercaSequenziale` è pari a $1+(2\mathtt{n}+1)+2 = 2\mathtt{n}+4$ dimostrando in questo modo che la complessità è proporzionale a n, dimensione dell'input.

1.2.3 Analisi asintotica e la notazione O

L'efficienza degli algoritmi impiegati ha grande rilevanza nel caso di input di dimensioni grandi. Infatti, nei casi di problemi di "piccola taglia," ovvero problemi i cui input hanno dimensione ridotta, la potenza di calcolo disponibile rende trascurabili inefficienze algoritmiche.

La situazione cambia radicalmente al crescere della dimensione dell'input. Ciò è molto ben illustrato dalla Tabella 1.2 (a). In tabella sono mostrate alcune possibili funzioni che esprimono un costo temporale di un esecutore in grado di eseguire una operazione in un microsecondo. Se nel futuro disporremo di sistemi mille volte più veloci, i tempi mostrati nella parte (a) si modificheranno dividendo per un fattore 1000 che rende comunque poco praticabile l'utilizzo di algoritmi con costo 2^n e 3^n (l'esecuzione per istanze di dimensioni 60 diverrebbe di 36,6 anni e 1.3×10^{10} secoli, rispettivamente).

nella parte b) della tabella confrontiamo algoritmi il cui aggiungiamo una costante moltiplicativa diversa a ciascun algoritmo.

Osserviamo che per piccole dimensioni tutti gli algoritmi hanno un costo pari al massimo ad un decimo di secondo e quindi le differenze fra i diversi algoritmi non hanno grande rilevanza pratica. Osserviamo che, quando le dimensioni dell'input questo non è più vero. Infatti anche solo per dimensioni pari a 60 gli algoritmi di costo $O(n^5)$ e quelli esponenziali sono molto peggiori degli altri: l'algoritmo di costo $O(n^5)$ ha un tempo di esecuzione più di 400 volte maggiore di quello $O(n^3)$ mentre gli algoritmi esponenziali hanno tempi di esecuzione non pratici. Inoltre osservaimo che l'algoritmo lineare è sicuramente quello da preferire avendo un tempo di esecuzione sessanta volte inferiore al migliore degli altri.

Se invece consideriamo la Tabella 1.2 (b) confrontiamo algoritmi in cui associamo una costante moltiplicativa diversa (decrescente sulle righe). Vediamo che per dimensione pari a 10 gli algoritmi esponenziali sono i migliori ma per dimensioni pari a 30 o maggiori il migliore algoritmo è quello lineare e gli algoritmi esponenziali sono molti ordini di grandezza peggiori degli altri,

In conclusione, *le tabelle evidenziano che l'intrattabilità computazionale è da associare all'andamento asintotico (esponenziale) della funzione di costo piuttosto che a costanti moltiplicative che a una possibile scarsa potenza di una piattaforma hardware/software.*

Inoltre la discussione precedente motiva come l'obiettivo primario dell'analisi del costo computazionale sia individuare l'andamento asintotico della funzione di costo. A tal fine il modello CEM e l'ipotesi semplificativa di assumere che il costo di esecuzione di ciascuna istruzione semplice sia unitario, per quanto rudimentale, consente di determinare l'andamento asintotico ottenendo una valutazione del costo di esecuzione di tipo matematico che non dipende dagli scenari di test adottati.

Infine ricordiamo che nell'analisi matematica due funzioni di variabile reale $f(x)$ e $g(x)$ hanno lo stesso andamento asintotico, e si scrive $f(x) \sim g(x)$, se

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 1$$

Nell'ambito dell'analisi degli algoritmi si usa il concetto lievemente differenziato "stesso andamento asintotico a meno di una costante," intendendo di fatto che il su indicato limite può avere qualunque risultato finito e positivo. Le funzioni impiegate sono di norma funzioni definite su $\mathbb{N}$ e a valori in $\mathbb{N}$.

Consideriamo nel seguito alcune proprietà delle notazioni introdotte.

Proprietà 1. $f(n) \in O(g(n)) \iff g(n) \in \Omega(f(n))$

Tabella 1.2 Confronto fra alcune funzioni polinomiali ed esponenziali di complessità temporale. In (a) si fa riferimento a un sistema di elaborazione in grado di eseguire 10^6 operazioni/s; in (b) si considerano le stesse funzioni di costo, ma in riferimento a un sistema di elaborazione in grado di eseguire 10^9 operazioni/s.

compl. temporale	10	20	30	40	50	60
n	10 μs	20 μs	30 μs	40 μs	50 μs	60 μs
n^2	0.1 ms	0.4 ms	0.9 ms	1.6 ms	2.5 ms	3.6 ms
n^3	1 ms	8 ms	27 ms	64 ms	125 ms	216 ms
n^5	0.1 s	3.2 s	24.3 s	1.7 min	5.2 min	13.0 min
2^n	1 ms	1 s	17.9 min	12.7 g	35.7 anni	366 secoli
3^n	59 ms	58 min	6.5 anni	3855 secoli	2×10^8 secoli	1.3×10^{13} secoli

(a)

compl. temporale	10	20	30	40	50	60
$10000n$	100 ms	200 ms	300 ms	400 ms	500 ms	600 ms
$1000n^2$	100 ms	400 ms	900 ms	1.6 s	2.5 s	3.6 s
$100n^3$	100 ms	800 ms	$2,7$ s	$6,4$ s	$12,5$ s	$21,6$ s
$10n^5$	10 s	32 s	243 s	17 min	52 min	130 min
2^n	1 ms	1 s	17.9 min	12.7 g	35.7 anni	366 secoli
3^n	59 ms	58 min	6.5 anni	3855 secoli	2×10^8 secoli	1.3×10^{13} secoli

(b)

Dimostrazione. ($\Longrightarrow$) Per ipotesi $f(n) \leq c \cdot g(n)$, per qualche costante positiva c ed n abbastanza grande. Allora $g(n) \geq f(n)/c = c' \cdot f(n)$, per n abbastanza grande e $c' = 1/c$, che, per definizione, significa che $g(n) \in \Omega(f(n))$.

($\Longleftarrow$) Simile al precedente e pertanto omesso, per brevità. □

Proprietà 2. $f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n)) \Longleftrightarrow f(n) \in \Theta(g(n))$

Dimostrazione. Discende immediatamente dalle definizioni delle notazioni utilizzate e viene omessa per brevità. □

Proprietà 3. $f(n) \in \Theta(g(n)) \Longleftrightarrow g(n) \in \Theta(f(n))$

Dimostrazione. ($\Longrightarrow$) Per ipotesi $c_1 g(n) \leq f(n) \leq c_2 g(n)$ per n definitivamente grande ed opportune costanti positive c_1 e c_2. Dividendo i tre membri delle disuguaglianze per c_1 otteniamo $g(n) \leq f(n)/c_1 \leq c_2 g(n)/c_1$ che possiamo riscrivere, ponendo $c_1' = 1/c_1$, come $g(n) \leq c_1' f(n) \leq c_1' c_2 g(n)$, dalla cui prima diseguaglianza si ottiene $g(n) \in O(f(n))$. Dividendo ora i tre membri di $c_1 g(n) \leq f(n) \leq c_2 g(n)$ per c_2 otteniamo $c_1 g(n)/c_2 \leq f(n)/c_2 \leq g(n)$ che possiamo riscrivere, ponendo $c_2' = 1/c_2$, come $c_1 c_2' g(n) \leq c_2' f(n) \leq g(n)$, dalla cui seconda diseguaglianza si ottiene $g(n) \in \Omega(f(n))$. Per la Proprietà 2 vale $g(n) \in \Theta(f(n))$.

($\Longleftarrow$) Simile al precedente e pertanto omesso, per brevità. □

Avendo dunque chiarito che l'obiettivo è quello di effettuare analisi asintotica, e cioè valutare l'andamento asintotico della funzione di costo (sia temporale che spaziale) dell'algoritmo, a meno di costanti moltiplicative (e additive), si può ribadire la comodità di impiego di una CEM per effettuare l'analisi, facilitati, per quanto appena discusso, dal fatto che le costanti moltiplicative/additive possono essere tralasciate durante questa analisi. Si noti che ciò vale anche per quanto riguarda la dimensione dell'input (cfr. **??**, da cui è anche ripresa la notazione k più avanti impiegata), giustificando pienamente il fatto che, quando l'input sia un array di n elementi, si utilizzi n, e non $k \cdot n$, come misura di tale dimensione. Nel seguito, ove non diversamente specificato, si adopererà la locuzione "stesso andamento asintotico" intendendo "stesso andamento asintotico a meno di costanti moltiplicative e/o additive." Egualmente per la locuzione "stesso costo asintotico."

1.2.4 Operazioni dominanti

Il concetto di operazione (o istruzione) dominate permette di semplificare significativamente la determinazione del costo computazionale (asintotico, di caso peggiore) in termini di tempo di un algoritmo.

Ciò premesso, una istruzione (o operazione) elementare σ di un programma linearizzato A è detta *dominante* se essa fornisce un contributo $C_\sigma(z)$ al costo $C_A(z)$ tale che esistono due costante positive ξ e ζ per cui, per ogni z:

$$\xi \cdot C_\sigma(z) + \zeta \geq C_A(z)$$

Da questa definizione, e dall'ovvia considerazione che $C_\sigma(z) \leq C_A(z)$, segue immediatamente che $C_\sigma(z)$ e $C_A(z)$ hanno lo stesso andamento asintotico.

Una proprietà particolarmente utile dell'istruzione dominante è quella di costituire l'istruzione più frequentemente[7] eseguita dall'algoritmo. A tal proposito, visto che la definizione di istruzione dominante è applicabile a un'istruzione elementare, è bene precisare che, qualora nell'algoritmo che si sta analizzando ci siano chiamate ad altri algoritmi, diviene necessario immaginare che alla chiamata venga sostituito il corpo dell'algoritmo chiamato, altrimenti si potrebbe andare incontro a valutazioni errate. Un esempio aiuterà a chiarire il concetto.

[7] Da un punto di vista asintotico; ad es., n esecuzioni e $n + k$ esecuzioni, essendo k una costante, sono (asintoticamente) equivalenti, nel computo dei costi.

```
...
for(i = 0; i < n; i++)          /* ciclo A */
    for(j = 0; j < n; j++)      /* ciclo B */
        a = 0;
...
for(k = 1; k <= n; k++)          /* ciclo C */
    b = f(k);
...
```

Dal frammento appare che l'istruzione `a = 0` è eseguita $\mathtt{n}^2$ volte, mentre `b = f(k)` (che non è elementare in quanto chiama la funzione `f(k)`) è eseguita `n` volte. Il costo da associare al ciclo A sarà $\mathtt{n}^2$. Se la funzione `f(k)` ha un costo pari a $\mathtt{k}^2$ ne segue che al ciclo C dovrà essere associato un costo evidentemente superiore, poiché pari a

$$\sum_{\mathtt{k}=1}^{\mathtt{n}} \mathtt{k}^2 = \frac{2\mathtt{n}^3 + 3\mathtt{n}^2 + \mathtt{n}}{6}$$

ove è stata sfruttata la formula per l'`n`-esimo numero piramidale quadrato,[8] che presenta lo stesso andamento di $\mathtt{n}^3$.

L'esempio mostra come, allo scopo di individuare correttamente un'istruzione dominante, sia opportuno "espandere" le chiamate di funzioni. Nel caso della funzione `f(k)`, solo la sua espansione può consentire di determinarne il costo $\mathtt{k}^2$. Considerazioni simili valgono nel caso di programmi ricorsivi.

In pratica, per individuare l'istruzione dominante, non è necessario applicare la sua definizione, ma basta riferirsi all'operazione più frequentemente eseguita dall'algoritmo (ad es., guardando all'interno dei cicli più interni). La componente di costo associata all'istruzione dominante determinerà l'andamento asintotico del costo temporale dell'algoritmo, e non sarà necessario conteggiare tutte le altre operazioni svolte. Da notare che l'istruzione più frequentemente eseguita è spesso non unica: in tal caso basterà prenderne in considerazione una qualunque che risponde a questa proprietà.

Le osservazioni e le considerazioni della sezione precedente permettono di esprimere il costo di esecuzione di un programma, o di un algoritmo, come una funzione delle dimensioni dell'ingresso in cui si trascurano le costanti moltiplicative e in cui siamo interessati al comportamento asintotico per grandi dimensioni dell'input.

Allo scopo di rendere più efficace, e precise, le valutazioni di andamento asintotico delle funzioni di costo si fa riferimento alla cosiddetta notazione asintotica. Data una funzione $g(n)$ $O(g(n))$ rappresenta l'insieme delle funzioni che - per grandi valori di n - crescono come $c \cdot g(n)$, dove c è una arbitaria costante. Formalemtne abbiamo la seguente definizione.

Data una funzione $g(n)$ definita su $\mathbb{N}$, $O(g(n))$ rappresenta l'insieme delle funzioni che per valori maggiori di n_0 assumono un valore pari al più a $c \cdot g(n)$; formalmente abbiamo:

$$O(g(n)) = \{f : \mathbb{N} \mapsto \mathbb{N} \mid (\exists c > 0)\ (\exists n_0 > 0)\ (\forall n \geq n_0)\ (f(n) \leq c \cdot g(n)\}$$

Se $f(n) \in O(g(n))$ si dice che "f cresce al più come g," per evidenziare che, asintoticamente, la crescita di f non eccede quella di g, a meno di costanti moltiplicative. Spesso, abusando della notazione, si scrive anche $f(n) = O(g(n))$. L'insieme delle funzioni costanti vieni in genere descritto con la notazione $O(1)$.

Ricordiamo che in analisi matematica due funzioni reali di variabile reale $f(x)$ e $g(x)$ hanno lo stesso andamento asintotico, e si scrive $f(x) \sim g(x)$, se

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = 1$$

[8] Si veda, ad es., la *On-Line Encyclopedia of Integer Sequences*, di Neil Sloane. `http://www.ams.org/notices/200308/comm-sloane.pdf`.

La definizione $O(g(n))$ usa il concetto simile di "stesso andamento asintotico a meno di una costante," intendendo di fatto che il su indicato limite può avere qualunque risultato finito e positivo.

Oltre alla notazione O-grande, sono utili le notazioni Ω-grande e Θ-grande, qui definite.

Data una funzione $g(n)$ definita su $\mathbb{N}$, definiamo $\Omega(g(n))$ come l'insieme delle funzioni che per valori maggiori di n_0 assumono un valore almeno pari a $c \cdot g(n)$; formalmente abbiamo:
$\Omega(g(n)) = \{f : \mathbb{N} \mapsto \mathbb{N} \mid (\exists c > 0)\ (\exists n_0 > 0)\ (\forall n \geq n_0)\ (f(n) \geq c \cdot g(n)\}$

Se $f(n) \in \Omega(g(n))$ si dice che "f cresce almeno come g"

Infine la notazione $\Theta(g(n))$ rappresenta l'insieme delle funzioni che sono sia $O(g(n))$ e $\Omega(n))$. Formalmente
$\Theta(g(n)) = \{f : \mathbb{N} \mapsto \mathbb{N} \mid (\exists c_1 > 0)\ (\exists c_2 > 0)\ (\exists n_0 > 0)\ (\forall n \geq n_0)\ (c_1 g(n) \leq f(n) \leq c_2 g(n)\}$

Se $f(n) \in \Theta(g(n))$ abbiamo che $f(n) \in O(g(n))$ e $f(n) \in \Omega(n))$ e si dice che "f e g crescono nello stesso modo."

Osserviamo che in base alla definzione se $f(n) \in O(n)$ allora abbiamo anche che $f(n) \in O(n^2)$ e $f(n) \in O(n^3$. Analogamente se $f(n) \in \Omega(n^2)$ allora abbiamo anche che $f(n) \in \Omega(n)$.

Nota. Molto spesso per denotare che $f(n) \in O(g(n))$ si usa la seguente notazione alternativa $f(n) = O(g(n))$ Usando l'eguale invece del simbolo di appartenenza è estremamente comune, ma è improprio scrivere $f(n) = O(g(n))$ per indicare che $f(n)$ appartenga all'insieme di funzioni $O(g(n))$.

È importante sottolineare indipendente dalla notazione usata che un'affermazione come $f(n) \in O(g(n))$ (o $f(n) = O(g(n))$) significa che f è "al massimo" g in un senso approssimato quando ignoriamo le costanti. Analogamente un'asserzione come $f(n) \in \Omega(g(n))$ (o $f(n) = \Omega(g(n))$) significa che f è 'almeno' g nello stesso senso approssimato.

Concludiamo presentando alcune semplici regole che possono essere d'aiuto quando si tenta di confrontare due funzioni f e g:

1. Le costanti moltiplicative e additive non contano nella notazione O; pertanto se $f(n) \in O(g(n))$ allora anche $(1000f(n) + 1000) \in O(g(n))$

2. Quando si completano le funzioni, è sufficiente che l'argomento sia più ampio. Ad esempio, ai fini della notazione, O ci interessa solo l'esponente più grande e pertanto, $f(n) = (n^3 + 100n^2 + 1000n) \in O(n^3)$

3. Date due costanti$a$ e b, $a \leq b$ allora abbiamo $n^a \in O(n^b)$ ma non è vero il contrario; pertanto $n^b \notin O(n^a)$

4. Un qualunque polinomio è sempre più piccolo di una funzione esponenziale; pertanto $n^a \in O(2^{nb})$ per ogni valore delle costanti a, $a > 0$ e b, $b > 0$ anche se b è molto più piccolo di a (abbiamo ad esempio $10n^{100} \in O(2^n)$ e $f(n) = 2^n \notin O(10n^{100})$

5. Analogamente una funzione logaritmica è sempre più piccola di una funzione polinomiale; pertanto abbiamo che $f(n) = 100n^2 \log(100n) \in O(n^3)$.

6. Nella maggioranza dei casi pratici (anche se non tutti!) in cui usiamo la notazione O le costanti nascoste non sono troppo grandi; in particolare a livello intuitivo, è ragionevole assumere che se $f(n) \in O(g(n))$ allora, per ogni n sufficientemente grande $f(n) \leq 100g(n) + 1000$. Una considerazione analoga vale nel caso della notazione Ω in cui possiamo assumere che le costanti nascostenon siano troppo piccole; pertanto se $f \in \Omega(g(n))$ allora nella maggioranza dei casi abbiamo che $f(n) \geq 0,001g(n)$ (sempre per ogni n sufficientemente grande).

   Questa osservazione giustifica il trascurare i fattori costanti moltiplicativie e additivi nella notazione O.

1.2.5 Composizione di algoritmi

Non è infrequente il caso in cui un algoritmo consiste nella composizione di due algoritmi. Più precisamente, dati due algoritmi A_1 e A_2, se l'output di A_1 e l'input di A_2 sono compatibili,

ovvero dello stesso tipo, è possibile definire un nuovo algoritmo A come illustrato in Fig. 1.2, ove si vede che l'input di A coincide con quello di A_1 e l'output di A coincide con quello di A_2, avendo usato l'output di A_1 come input di A_2. Se precedenti analisi hanno permesso di ricavare i costi

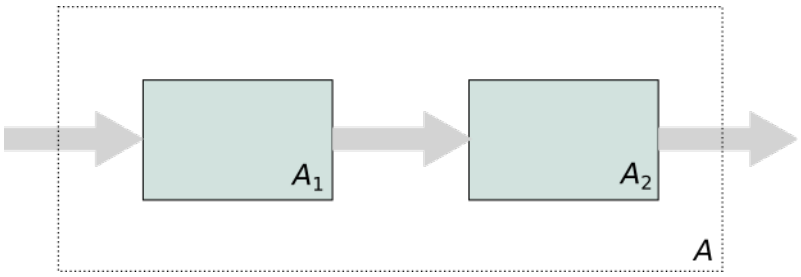

Figura 1.2 Composizione degli algoritmi A_1 e A_2 per definire un nuovo algoritmo A.

di A_1 e A_2, siano essi rispettivamente $C_1(n)$ e $C_2(n)$, allora è facile ricavare il costo dell'intero algoritmo A. Basta sommare le due componenti di costo, avendo cura di tener conto che, se l'input di A_1 ha dimensione n, non è assolutamente ovvio che l'input di A_2 abbia dimensione n, per cui sarebbe errato pensare che il costo di A, denotato con C, sia una funzione semplicemente ottenuta sommando algebricamente le due funzioni di costo C_1 e C_2. In altre parole, in generale avremo che $C(n) \neq C_1(n) + C_2(n)$. Va infatti tenuto conto della possibile diversa dimensione dell'input di A_2. A tale scopo, notando che l'input di A_2 altri non è che l'output di A_1, diviene utile far riferimento al coefficiente di dilatazione (massima) introdotto nella Sezione **??**. Se denotiamo con x l'input di A_1, con $|x| = n$, avremo che la dimensione dell'input di A_2 sarà $n \cdot \delta_{A_1}(n)$, valore da usare per determinare la somma corretta dei costi: $C(n) = C_1(n) + C_2(n \cdot \delta_{A_1}(n))$.

È immediato verificare che nella composizione di tre algoritmi A_1, A_2 e A_3, di costi, rispettivamente, C_1, C_2 e C_3, il costo totale sarà

$$C(n) = C_1(n) + C_2(n\,\delta_{A_1}(n)) + C_3(n\,\delta_{A_1}(n)\,\delta_{A_2}(n\,\delta_{A_1}(n)))$$

La generalizzazione al caso di k algoritmi è ovvia.

1.3 Esempi di analisi di complessità

In questa sezione vediamo diversi esempi di analisi di complessità.

1.3.1 Ricerca binaria

Consideriamo ora il seguente algoritmo di Ricerca binaria.

```
11. /*
12.   restituisce indice della posizione di k
13.   oppure -1 se k non presente; assume n > 0
14. */
15. int ricercaBinaria(int a[], int n, int k) {
16.   int primo = 0, ultimo = n - 1, media;
17.   while(primo <= ultimo) {
18.     media = (primo + ultimo) / 2;
19.     if(a[media] == k) return media;
20.     else if(a[media] < k) primo = media + 1;
21.     else ultimo = media - 1;
22.    }
23.   return -1;
24. }
```

Per quanto riguarda l'algoritmo di ricerca binaria abbiamo che:

- l'istruzione condizionale nelle linee `19`–`21.` ha costo massimo pari a 3 (nel percorso più lungo occorre valutare due espressioni semplici ed eseguire un'assegnazione);
- per il ciclo `while` alle linee `17`–`22.` indichiamo con $q \geq 0$ il numero di iterazioni eseguite: l'espressione che controlla il `while` è semplice e ha costo unitario, il corpo del ciclo ha un costo massimo pari a $1+3=4$, quindi il costo di q iterazioni sarà pari a $(q+1)+4q = 5q+1$.

  Per determinare il massimo valore possibile per q basta osservare che ad ogni iterazione la differenza `ultimo` − `primo` si dimezza,[9] per cui dovendo essere `primo` ≤ `ultimo`, e vista la loro inizializzazione, si avrà che il massimo numero di dimezzamenti sarà pari a $\lfloor \log_2 \mathtt{n} \rfloor + 1$, che è anche il massimo valore di q; ne deriva che il costo massimo del ciclo `while` sarà pari a $5 \lfloor \log_2 \mathtt{n} \rfloor + 6$;

  Il costo di esecuzione della procedura di ricerca binaria dipende dal particolare elemento cercato; se esso coincide con l'elemento mediano della tavola, allora è sufficiente un solo confronto; in tutti gli altri casi dobbiamo eseguire un numero maggiore di confronti. Per valutare la complessità della procedura di ricerca binaria nel caso peggiore osserviamo che, dopo un confronto, dobbiamo proseguire la ricerca su metà degli elementi. Inoltre, ogni successivo confronto permette di dimezzare ulteriormente la dimensione della tavola su cui proseguire la ricerca. Pertanto, il secondo confronto riduce la dimensione della tavola su cui dobbiamo effettuare la ricerca ad un quarto degli elementi della tavola (la metà della metà), il terzo confronto ad un ottavo e così via. Dopo m confronti la dimensione della tavola su cui dobbiamo continuare la ricerca è

  $$\frac{n}{2^m}$$

  La ricerca prosegue fino a quando non si trova l'elemento cercato oppure la dimensione della tavola in cui dobbiamo ancora cercare è uno; in questo caso un ulteriore confronto permette di individuare l'elemento cercato o di stabilire che esso non esiste. Nel caso peggiore, il numero di confronti è il più piccolo intero m che tale che

  $$\frac{n}{2^m} \leq 1 \text{ cioè } n \leq 2^m.$$

  Facendo il logaritmo si ottiene

  $$\log_2 n \leq \log_2 2^m \text{ cioè } m \geq \log_2 n.$$

  Pertanto, non appena m è maggiore o uguale a $\log_2 n$ allora la ricerca si arresta. Abbiamo, quindi, dimostrato che nel caso peggiore, l'algoritmo di ricerca binaria ha complessità $O(\log_2 n)$ ovvero che *la complessità dell'algoritmo (o del programma di ricerca binaria) in una tavola ordinata con n elementi è* $O(\log_2 n)$.
- il `return` a linea `23.` ha costo unitario.

Sulla base di questa discussione si può dunque concludere che su una CEM il costo temporale massimo di Ricerca Binaria è pari a

$$2 + (5 \lfloor \log_2 \mathtt{n} \rfloor + 6) + 1 = 5 \lfloor \log_2 \mathtt{n} \rfloor + 9 = O(\log_2 n)$$

Il confronto fra i due algoritmi mostra una sostanziale equivalenza in termini di costo spaziale (due variabili intere, ovvero una manciata di byte in più, impiegati da `ricercaBinaria`) e un salto esponenziale fra le due funzioni per quanto concerne il loro costo temporale: andamento logaritmico in `n` per `ricercaBinaria`, contro un andamento lineare in `n` per `ricercaSequenziale`.

Per quanto riguarda il costo spaziale osserviamo che `ricercaSequenziale` impiega un array di `n` interi e tre variabili intere, vale a dire lo spazio di `n` + 3 variabili intere; `ricercaBinaria` impiega un array di `n` interi e cinque variabili intere, vale a dire lo spazio di `n` + 5 variabili intere. Quindi ambedue gli algoritmi richiedono spazio $O(n)$ proporzionale alla dimensione dell'input.

[9] Per la precisione, la differenza massima si aggiorna con la legge $d_{i+1} = \lfloor (d_i - 1)/2 \rfloor$.

(a)		(b)		(c)		(d)
3		3		3	$\leftarrow j-1$	2
6		6	$\leftarrow j-1$	2	$\leftarrow j$	3
4	$\leftarrow j-1$	2	$\leftarrow j$	6		6
2	$\leftarrow j$	4		4		4

(e)		(f)		(g)		(h)
2		2		2		2
3		3	$\leftarrow j-1$	3		3
6	$\leftarrow j-1$	4	$\leftarrow j$	4	$\leftarrow j-1$	4
4	$\leftarrow j$	6		6	$\leftarrow j$	6

Figura 1.3 Esecuzione dell'algoritmo di ordinamento a bolle

.

1.3.2 Ordinamento a bolle (bubble sort)

L'algoritmo di ordinamento a bolle cerca di trarre vantaggio da un eventuale ordinamento parziale del vettore. Vediamo prima una versione semplificata dell'algoritmo che risulta meno efficiente della versione definitiva, ma che è più utile per presentare l'idea fondamentale. L'algoritmo nella sua versione semplificata consiste di n fasi.

1. La prima fase parte da $A[n]$ e inizia una serie di confronti e scambi:

   (a) confronta $A[n]$ e $A[n-1]$; se $A[n] < A[n-1]$, allora i due elementi non rispettano l'ordinamento e vengono scambiati; invece se $A[n] \geq A[n-1]$, allora i due elementi rispettano l'ordinamento crescente e non si fa nulla;

   (b) l'algoritmo prosegue confrontando le componenti $A[n-1]$ e $A[n-2]$ e scambiandole fra loro se $A[n-1] < A[n-2]$;

   (c) si prosegue confrontando ed eventualmente scambiando le componenti $A[n-2]$ e $A[n-3]$;

   (d) poi $A[n-3]$ e $A[n-4]$, e così via fino a quando non si esaminano $A[2]$ e $A[1]$ che vengono eventualmente scambiate.

   Termina in questo modo la prima fase dell'algoritmo. Alla fine della prima fase l'elemento più piccolo è risalito fino alla prima posizione.

2. La seconda fase dell'algoritmo è analoga:

   (a) si esaminano le componenti $A[n]$ e $A[n-1]$ e si scambiano fra loro se non verificano l'ordinamento desiderato;

   (b) si prosegue confrontando (ed eventualmente scambiando) $A[n-1]$ e $A[n-2]$;

   (c) si confronta $A[n-2]$ e $A[n-3]$ ecc., fino a quando non si confrontano (ed eventualmente si scambiano) $A[3]$ e $A[2]$.

   A questo punto termina la seconda fase e la seconda componente più piccola dell'array è stata sistemata in seconda posizione.

$\vdots$

(i) Le fasi successive proseguono nello stesso modo; pertanto, al termine della fase i, l'algoritmo ha posto l'i-esimo più piccolo elemento dell'array in posizione i.

Se immaginiamo di associare a ciascun elemento del vettore un peso pari al suo valore, allora possiamo dire che gli elementi più leggeri sono risaliti fino in cima come bolle; da questa similitudine deriva il nome di *ordinamento a bolle* dell'algoritmo.

Il programma che lo implementa è il seguente:

(a)	(b)	(c)	(d)	(e)	(f)
6	1	1	1	1	1
5	6	2	2	2	2
4	5	6	3	3	3
3	4	5	6	4	4
2	3	4	5	6	5
1	2	3	4	5	6

Figura 1.4 Comportamento più sfavorevole dell'algoritmo di ordinamento a bolle

Esempio 2. Un esempio di ordinamento a bolle di un vettore è mostrato in figura 1.3.

All'inizio si confrontano fra loro la quarta e la terza componente. Poiché $A[4]$ è minore di $A[3]$, si scambiano fra loro i valori delle due componenti ottenendo il vettore di figura 1.3-b. Successivamente si confrontano e si scambiano $A[3]$ e $A[2]$ (vedi figura 1.3-c), poi $A[2]$ e $A[1]$ ottenendo alla fine la situazione di figura 1.3-d. Durante la fase successiva otteniamo via via i vettori di figura 1.3-e, f, g. Durante l'ulteriore fase, si confrontano nuovamente $A[4]$ e $A[3]$ (vedi figura 1.3-h); in questa fase non avverrà nessuno scambio, perché il vettore è già ordinato. □

Esercizio

Si analizzi la complessità dell'algoritmo di ordinamento a bolle in versione semplificata.

Per cercare di migliorare l'algoritmo di ordinamento a bolle è sufficiente osservare che non sempre sono necessarie $n-1$ fasi, ma può accadere che il vettore sia stato ordinato prima che l'ultima fase sia completata. In particolare, osserviamo che

- se durante una fase non avviene nessuno scambio allora il vettore è ordinato;
- se durante una fase avviene almeno uno scambio allora non possiamo sapere se il vettore sia stato ordinato o meno.

La seconda versione dell'algoritmo di ordinamento a bolle utilizza l'osservazione precedente introducendo una variabile booleana `ordinato` che all'inizio di ogni fase viene posta uguale a `true` e, se avviene uno scambio durante la fase, viene resa pari a `false`. In questo modo, alla fine di una fase, la variabile `ordinato` è vera se non è avvenuto nessuno scambio ed è falsa altrimenti.

Il costo di esecuzione dell'algoritmo precedente dipende dalla configurazione dei dati di ingresso. Supponiamo infatti che l'array di ingresso sia già ordinato in senso crescente. In questo caso l'algoritmo compie una sola scansione dell'array durante la quale non si compie nessuno scambio, ma che serve a verificare che l'array sia effettivamente ordinato in senso crescente. Pertanto, in questo caso il costo di esecuzione dell'algoritmo è $O(n)$. Valutiamo ora la complessità dell'algoritmo di ordinamento a bolle nel caso peggiore. Il caso peggiore dell'algoritmo è quello in cui sono necessarie $n-1$ fasi per completare la sua esecuzione. Questa condizione si verifica quando il vettore è ordinato in senso decrescente e deve essere ordinato in senso crescente.

Consideriamo ad esempio il vettore nella colonna (a) della figura 1.4. Nella colonna (b) della figura 1.4 è dato il vettore alla fine della prima fase, durante la quale l'elemento più piccolo è posto in prima posizione. Nelle colonne (c), (d), (e) ed (f) viene dato il vettore alla fine delle fasi successive. Si noti che alla fine di ogni fase un solo elemento del vettore è stato posto nella posizione definitiva. Pertanto, per ordinare un vettore di n elementi possono essere necessarie $n-1$ fasi. Il numero dei confronti effettuati per la prima fase è $n-1$, per la seconda fase è $n-2$ e per la fase i è $n-i$. Pertanto il numero totale di confronti è

$$(n-1)+(n-2)+(n-3)+\cdots+3+2+1 \;=\; \frac{n\times(n-1)}{2}$$

Otteniamo quindi che *la complessità dell'algoritmo per l'ordinamento a bolle ottimizzato è* $O(n^2)$.

La Tabella 1.3 riporta i costi asintotici di caso peggiore degli algoritmi precedentemente discussi.

La Tabella 1.4 elenca le istruzioni dominanti presenti negli algoritmi mostrati nelle sezioni precedenti.

Tabella 1.3 Costi asintotici di alcuni algoritmi. Si è usato il simbolo z per denotare la dimensione dell'input. Si noti che nell'esprimere un andamento asintotico di tipo logaritmico non è necessario specificare la base del logaritmo, poiché un cambio di base equivale a moltiplicare per una costante.

algoritmo	costo asintotico
`ricercaSequenziale`	$\Theta(z)$
`ricercaBinaria`	$\Theta(\log z)$
`potenza`	$\Theta(2^z)$
`primo`	$\Theta(2^{z/2})$
`ordinamento a bolle`	$\Theta(n^2)$

Tabella 1.4 Istruzioni dominanti di alcuni algoritmi.

algoritmo	istruz. dominante	esecuzioni	linea
`ricercaSequenziale(a,n,k)`	`(i < n) && (a[i] < k)`	$\mathtt{n}+1$	7.
	`i=i+1`	`n`	7.
`ricercaBinaria(a,n,k)`	`primo <= ultimo`	$\lfloor\log_2 \mathtt{n}\rfloor+2$	17.
	`media=(primo+ultimo)/2`	$\lfloor\log_2 \mathtt{n}\rfloor+1$	18.
	`if(a[media] == k)`	$\lfloor\log_2 \mathtt{n}\rfloor+1$	19.
	`if(a[media] < k)`	$\lfloor\log_2 \mathtt{n}\rfloor+1$	20.
	`primo = media + 1`	$\lfloor\log_2 \mathtt{n}\rfloor+1$	20.
	`ultimo = media - 1`	$\lfloor\log_2 \mathtt{n}\rfloor+1$	21.
`potenza(a, b)`	`b > 0`	$\mathtt{b}+1$	31.
	`f = f * a`	`b`	32.
	`b = b - 1`	`b`	33.
`primo(n)`	`d <= sqrt(n)`	$\lceil\lfloor\sqrt{\mathtt{n}}\rfloor/2\rceil$	46.
	`d += 2`	$\lceil\lfloor\sqrt{\mathtt{n}}\rfloor/2\rceil-1$	46.
	`if(!(n % d))`	$\lceil\lfloor\sqrt{\mathtt{n}}\rfloor/2\rceil-1$	47.
`????`	`???  a[i]=potenza(a[i],exp)`	`n`	60.

1.3.3 Calcolo della potenza di un numero

ALB: questo esempio ha il problema che possiamo andare in overflow; inserire un commento bisogna precisarlo

Il seguente esempio ci permette di evidenziare l'importanza di saper riconoscere quale parte dell'input è davvero rilevante ai fini della modellazione del costo. Ciò verrà meglio illustrato attraverso un altro esempio.

```
25. /*
26.    determina la potenza a^b
27.    assume b >= 0
28. */
29. int potenza(int a, int b) {
30.   int f = 1;
31.   while(b > 0) {
32.     f = f * a;
33.     b = b - 1;
34.   }
35.   return f;
36. }
```

L'algoritmo `potenza` determina il valore $\mathtt{a}^{\mathtt{b}}$ sostanzialmente attraverso `b` moltiplicazioni. Il costo dell'algoritmo su una CEM è facilmente determinato: costi unitari per le assegnazioni nelle

linee `30.`, `32.` e `33.`, costo del ciclo `while` nelle linee `31-34.` pari a $3\mathtt{b}+1$ (il ciclo prevede `b` iterazioni, per cui $\mathtt{b}+1$ test di controllo e $2\mathtt{b}$ assegnazioni) e costo unitario per il `return` a linea `35.`, per un totale pari a $3\mathtt{b}+3$.

Si può rilevare che il costo dell'algoritmo dipende solo da uno dei due interi forniti in input all'algoritmo. In particolare, all'aumento di `b` corrisponderà un maggiore tempo di esecuzione, mentre non si avranno variazioni apprezzabili del tempo di esecuzione al variare di `a`. In altre parole, può accadere che il tempo di esecuzione non dipenda da tutto l'input, ma solo da una sua parte. Sebbene ciò sia facilmente rilevabile *a posteriori*, l'analista esperto sarà capace di identificare *a priori* eventuali componenti di input non significative ai fini dell'analisi di costo, potendole dunque immediatamente tralasciare.

Si noti che nell'esempio appena esaminato la funzione determinata $3\mathtt{b}+3$ non descrive il costo in funzione della dimensione dell'input, poiché questa è funzione di *b valore dell'input* `b` e *non della dimensione dell'input pari a* $|\mathtt{b}|$.

Denotando la dimensione dell'input con z, si ha che utilizzando la notazione usuale per rappresentare i numeri labbiamo che $z = |\mathtt{b}| = \lfloor \log_2 \mathtt{b} \rfloor + 1$. Osserviamo che $z = |\mathtt{b}| = \lfloor \log_2 \mathtt{b} \rfloor + 1$ implica che $b = O(2^{\mathtt{b}}$ e quindi la complessità finale del programma è $O(2^{\mathtt{b}}$, una funzione esponenziale.

1.3.4 Test di primalità

Presentiamo ora un algoritmo che effettua il test di primalità con lo scopo di chiarire ulteriormente la differenza fra dimensione e valore dell'input, [10] su un intero positivo `n`. L'algoritmo non verifica esplicitamente tutti i possibili divisori ma si limita a considerare l'insieme $\{2\} \cup \{3, 5, 7, 9, \ldots, \lfloor \sqrt{\mathtt{n}} \rfloor\}$ che contiene 2 e i numeri dispari inferiori a $\sqrt{\mathtt{n}}$. Infatti se non esiste un divisore minore o uguale della radice quadrata del numero allora non esiste divisore.

Se il numero in input è primo l'algortimo restituisce 1 altrimenti restituisce un divisore del numero dato.

```
37.  int primo(int n) {
38.   /*
39.   restituisce 1 se n è primo
40.   oppure un divisore <= sqrt(n)
41.   assume n > 1
41. */
43.   int d;
44.   if(n <= 3) return 1;              /* ritorna se n è 2 o 3 */
45.   if(!(n % 2)) return 2;            /* ritorna se n è pari */
46.   for(d = 3; d <= sqrt(n); d += 2)  /* inutile testare numeri pari */
47.     if(!(n % d))
48.       return d;                     /* n è divisibile per d */
49.   return 1;                         /* n è primo */
50. }
```

È evidente che l'algoritmo `primo` massimizza la lunghezza della sua computazione quando l'input è un numero primo maggiore di 3, terminando con il `return 1` di linea `49.` Il costo è in tal caso pari a $3 + (p+1) + p + p + 1 = 3p + 5$ essendo p il numero massimo di iterazioni del ciclo `for`. Alternativamente osserviamo che nel caso di numero primo maggiore di 3 le istruzioni 46. e 47. sono istruzioni dominanti eseguite p volte.

Per valutare p osservaimo che sono provati tutti i numeri dispari compresi fra 3 e $\sqrt{\mathtt{n}}$ e quindi abbiamo

$$p = \left\lceil \left\lfloor \sqrt{\mathtt{n}} \right\rfloor / 2 \right\rceil - 1 \approx \sqrt{\mathtt{n}}/2$$

Pertanto, il costo in funzione di `n` è $O(\sqrt{\mathtt{n}}/2)$. Per esprimerlo in funzione di $z = |\mathtt{n}|$ basta rammentare che $|\mathtt{n}| = \lfloor \log_2 \mathtt{n} \rfloor + 1 \approx \log_2 \mathtt{n} + 1$, da cui $\mathtt{n} \approx 2^{z-1}$ e $\sqrt{\mathtt{n}} \approx 2^{\frac{z-1}{2}}$, ottenendo

[10] Un intero $\mathtt{n} > 1$ è *primo* se ammette i due soli divisori 1 e `n`.

dunque un costo in funzione della dimensione dell'input pari a $O(2^{\frac{z-3}{2}})$, che manifesta dunque un andamento esponenziale.

Anche in questo esempio, così come in `potenza`, esprimere il costo in funzione del valore dell'input (`n`) piuttosto che in funzione della sua dimensione (|`n`|) modifica profondamente l'espressione della funzione di costo (sub-lineare contro esponenziale), per cui è in ogni caso critico, ai fini di un'analisi dei costi corretta e standardizzata, evitare la confusione fra i due concetti (valore e dimensione).

1.4 Analisi di algoritmi ricorsivi

Valutare la complessità di un algoritmo ricorsivo è, in genere, più complesso che nel caso degli algoritmi iterativi. Infatti, la natura ricorsiva della soluzione algoritmica dà luogo a una funzione di costo che, essendo strettamente legata alla struttura dell'algoritmo, è anch'essa ricorsiva. Questa equazione - detta *equazione di ricorrenza* - è una equazione che definisce una sequenza di valori a partire da uno o più valori iniziali dati; ogni ulteriore valore della funzione è definito come funzione dei valori precedenti. Vediamo ad esempio la funzione di ricorrenza così definita:

$G(n) = 1$ se $n = 1$ e $G(n) = 2G(n-1)$ se $n > 1$

La definizione precedente fornisce un solo valore iniziale ($G(1)$) e definisce gli altri termini in funzione dei precedenti. Ad esempio per calcolare $G(3)$ dobbiamo prima calcolare $G(2)$ il cui valore è calcolabile in base alla definizione come due volte il valore di $G(1)$. Ne cosnegue che $G(3)$ è pari a 4. (In questo caso non è difficile verificare che i valori di $G(n)$ sono $1, 2, 4, 8, 16 \ldots$ le potenze di 2.

Dato un algoritmo in genere, trovare la funzione di costo ricorsiva è piuttosto immediato. La parte meno semplice è la soluzione dell'equazione. La riformulazione della funzione di costo ricorsiva in una equivalente ma non ricorsiva si affronta impostando una equazione di ricorrenza, costituita dalla formulazione ricorsiva e dal caso base.

1.4.1 Equazioni di ricorrenza

Iniziamo da un esempio, quello della ricerca binaria ricorsiva. Il corpo della funzione, chiamato su un vettore di dimensione n:

- esegue un test verificando se l'elemento cercato è presente nel vettore
- se tale test non è soddisfatto effettua una chiamata ricorsiva su un vettore di dimensioni $n/2$ elementi.

Indicando con $T(n)$ la complessità dell'algoritmo ricorsivo, possiamo esprimere la complessità dell'algoritmo mediante questa equazione di ricorrenza:

- $T(n) = T(n/2) + \Theta(1)$
- $T(1) = \Theta(1)$

La soluzione delle equazioni di ricorrenza può essere fatta in diversi modi. Noi utilizzeremo il metodo più intuitivo e semplice basato sullo srotolamento

$$\begin{aligned} T(n) &= 1 + T(n/2) \\ &= 1 + 1 + T(n/4) = 2 + T(n/4) \\ &= 2 + 1 + T(n/8) = 3 + T(n/8) \\ &= 3 + 1 + T(n/16) = 4 + T(n/16) \\ &= \ldots \\ &= k + T(n/2^k) \end{aligned}$$

Assumiamo che n sia potenza di 2. In questo caso continuiamo a srotolare la ricorsione fin quando $n/2^k = 1$; ora $n/2^k = 1$ implica $2^k = n$ e quindi $k = log_2 n$ (k è il logaritmo in base 2 di n). Se n non è potenza di due allora interrompiamo lo srotolamento non appena $n/2^k < 1$. Questo implica che $n/2^{k-1} < 2$ e quindi $k = log_2 n + 1$. Abbiamo quindi mostrato che la soluzione dell'equazione nel caso della ricerca binaria è

$$T(n) = log_2 n + 1 = O(log_2 n)$$

1.4.2 Ordinamento per fusione (Mergesort)

Consideriamo ora l'algoritmo di ordinamento per fusione, detto Mergesort, un algoritmo ricorsivo che, nel caso peggiore, ha costo $O(nlogn)$ per ordinare una collezione di n elementi.

Dato un array di n elementi se se $n = 1$ allora l'array è ovviamente ordinato e l'algoritmo termina; altrimenti, se $n > 1$, si eseguono i seguenti passi:

1. Ordina la prima metà dell'array (in particolare gli elementi $\{1, 2, \ldots, \lceil \frac{n}{2} \rceil\}$)
2. Ordina la seconda metà dell'array (in particolare gli elementi $\{\lceil \frac{n}{2} \rceil + 1, \lceil \frac{n}{2} \rceil + 2, \ldots, n\}$)
3. Unisci (merge) le due parti ordinate in un unico array ordinato

Nel Seguito è illustrato un programma che lo realizza.

```
void merge(int low, int mid, int high) {
   int l1, l2, i;

   for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {
      if(a[l1] <= a[l2])
         b[i] = a[l1++];
      else
         b[i] = a[l2++];
   }
   while(l1 <= mid)
      b[i++] = a[l1++];
   while(l2 <= high)
      b[i++] = a[l2++];
   for(i = low; i <= high; i++)
      a[i] = b[i];
}

void sort(int low, int high) {
   int mid;

   if(low < high) {
      mid = (low + high) / 2;
      sort(low, mid);
      sort(mid+1, high);
      merging(low, mid, high);
   } else {
      return;
   }
}

int main() {
   int i;
```

```
   printf("List before sorting\n");
   for(i = 0; i <= max; i++)
      printf("%d ", a[i]);
   sort(0, max);
   printf("\nList after sorting\n");
   for(i = 0; i <= max; i++)
      printf("%d ", a[i]);
}
```

Osserviamo che il costo di merge è lineare nelle dimensioni. Infatti ogni volta che si esegue un ciclo almeno un elemento è inserito nell'array.

In questo caso l'equazione di ricorrenza è la seguente

$T(n) = 1$ se $n = 1$

$T(n) = n + 2T(n/2)$ se $n > 1$

Applicando il metodo dello srotolamento e assumendo per semplicità che n sia potenza di 2 e che la fusione costi esattamente n abbiamo

$$\begin{aligned} T(n) &= n + 2T(n/2) \\ &= n + 2(n/2 + 2T(n/4)) = 2n + 4T(n/4) \\ &= 2n + 4(n/4 + 8T(n/8)) = 3n + 8T(n/8) \\ &\dots \\ &= kn + 2^k(n/2^k) \end{aligned}$$

Il valore finale di k da considerare é quello per cui $n/2^k = 1$ e, quindi $k = \log_2 n$. Ponendo $k = \log_2 n$ nell'equazione precedente otteniamo che la soluzione dell'equazione di ricorrenza per il programma di ordinamento per fusione è pari a $n \log_2 n + n$.

Vediamo ora cosa succede rimuovendo separatamente le due ipotesi semplificative precedenti.

- n **non è potenza di due** In questo caso sia n' la più piccola potenza di 2 maggiore di n e sia z un valore minore al più piccolo elemento presente nell'array. Ovviamente abbiamo $n' < 2n$. Supponiamo ora di ordinare un array diinput nuovo ottenuto da quello dato in input aggiungendo $n' - n$ elementi pari a z. Osserviamo che il costo di esecuzione dell'algoritmo di fusione su un array di dimensione n' è non inferiore al costo di esecuzione su un array di dimensione n.

  In base a quanto abbiamo discusso precedentmente il tempo di esecuzione dell'algoritmo di fusione al nuovo input è $O(n' log_2 n')$. Dato che $n' < 2n$ abbiamo che $n' \log_2 n' < 2n(\log_2 n + 1)$. Quindi concludiamo che il costo di esecuzione dell'algoritmo di ordinamento per fusione di un array di dimensione n è $O(2n(\log_2 n + 1)) = O(n \log_2 n)$.

- **il costo di fusione non è** n
  In questo caso osserviamo che il costo di fusione lineare in n; quindi esiste una costante c tale che il costo di fusione è al massimo cn. Abbiamo

$$\begin{aligned} T(n) &= cn + 2T(n/2) \\ &= cn + 2(cn/2 + 2T(n/4)) = 2n + 4T(n/4) \\ &= 2cn + 4(cn/4 + 8T(n/8)) = 3n + 8T(n/8) \\ &\dots \\ &= kcn + 2^k(n/2^k) \end{aligned}$$

Non è difficile mostrare che quando ambedue le ipotesi semplificative sono rimosse otteniamo lo stesso risultato.

Osserviamo inoltre che Mergesort nel caso peggiore ha costo $\Omega(n \log n)$. Infatti l'algoritmo ha un unico test sulle dimensioni dell'array. Pertanto esegue le due attivazioni ricorsive e il merge sono eseguiti in ogni caso. COncludiamo quindi che il costo di Mergesort nel caso peggiore è anch'esso $\Omega(n \log n)$.

1.4.3 Calcolo del Massimo Comun Divisore, l'algoritmo di Euclide

Consideriamo il problema di decidere se due numeri interi a e b siano relativamente primi, ovvero se il loro massimo comun divisore (indicato con $\text{MCD}(a, b)$) sia uguale a 1. Un algoritmo con complessità temporale polinomiale per la risoluzione di tale problema, noto sotto il nome di algoritmo di Euclide, si basa sulla seguente relazione (la cui dimostrazione è lasciata come esercizio).

Dati due numeri interi nonengativi x e y assumiamo che $x \geq y$.

La definizione di MCD implica immeditamente che se $y = 0$ allora $\text{MCD}(x, 0) = x$. Se invece y non è zero possiamo dividere x per y; ricordando che $x \geq y$ possiamo scrivere

$x = yq + r$

La definizione di MCD implica che $\text{MCD}(x, y) = \text{MCD}(y, r)$ dove r è il resto della divisione che è ovviamente minore di y.

Ricordando che in C `(x % y)` denota il resto della divisione di x per y otteniamo il seguente programma.

```
int mcd_algorithm(int x, int y) {
/*
   programma per il calcolo
   del Massim Comun Divisore
   assume x >= y
/*
    if (y == 0) return x;
    else return mcd_algorithm(y, (x % y));
}
```

Per valutare la complessità temporale di tale algoritmo, mostriamo che sono sufficienti $O(\log b)$ chiamate ricorsive, avendo indicato con b il valore del parametro y alla prima chiamata. Siano (a_{k-1}, b_{k-1}), (a_k, b_k) e (a_{k+1}, b_{k+1}) tre coppie successive di valori su cui viene invocata la funzione `MCD`. Osserviamo innanzitutto che $a_k = qb_k + b_{k+1}$ dove q è intero positivo e, quindi, $b_{k-1} > a_k$. Dimostriamo ora che $b_{k-1} > b_k + b_{k+1}$. In effetti, abbiamo che $a_k = qb_k + b_{k+1}$ per cui $a_k > b_k + b_{k+1}$; dato che $b_{k-1} > a_k$ segue che $b_{k-1} > b_k + b_{k+1}$.

Osserviamo ora che $b_{k-1} > b_k + b_{k+1}$ e il fatto che $b_{k-1} > b_k > b_{k+1}$ implica che $b_{k+1} < b_{k-1}/2$. Osserviamo ora che i numeri sono interi e, quindi, il valore dimezza ogni due iterazioni, diventa un quarto ogni quattro iterazioni, diventa un sedicesimo ogni sei iterazioni e così via. In particolare, ogni $2k$ iterazioni il valore di b diventa una frazione pari a $1/2^k$. Questo implica che il numero di attivazioni complessive è al massimo $2 \log b = O(\log b)$.

Poiché ciascuna chiamata ricorsiva richiede tempo costante possiamo quindi affermare che la complessità dell'algoritmo di Euclide è $O(\log b)$. Poiché per rappresentare i valori di ingresso a e b usiamo $\log a$ e $\log b$ bit, possiamo concludere che l'algoritmo di Euclide ha complessità lineare nella dimensione dell'input.

1.4.4 Calcolo dei numeri di Fibonacci

Questo esempio finale è utile per illustrare icome una diretta realizzazione di un algoritmo ricorsivo possa avere un costo (sia in temrini di tempo che di memoria) non accettabile.

I numeri di Fibonacci sono così definiti: $F(n)$ il valore della funzione di FIbonacci è definito dalla seguente equazione di ricorrenza

$F(n) = 1$ `sen` < 3

$F(n) = F(n-1) + F(n-2)$ `sen` ≥ 3

Una diretta implementazione della definizione porta al seguente programma ricorsivo

```
def F(n)
iif(N==0 || N==1) fib = 1;
else{
F(n) = F(n-1)+ F(n-2)
}
```

Analizziamo ora il costo del programma precedente. Osserviamo che il costo di attivazione di $F(n)$ è costante più il costo delle eventuali attivazioni ricorsive. Quindi in base ai ragionamenti fatti possiamo affermare che il costo di $F(n)$, a meno di costanti moltiplicative e additive è pari a 1 più il numero di attivazioni ricorsive della funzione.

Consideriamo la figura seguente che mostra l'esecuzione del programma per $n = 6$. In particolare quando $n = 6$ sono chiamate ricorsivamente le funzioni $F(5)$ e $F(4)$ per i valori $n = 5$ e $n = 4$. L'osservazione cruciale è che $F(5)$ richiede una *nuova* attivazione di $F(4)$!

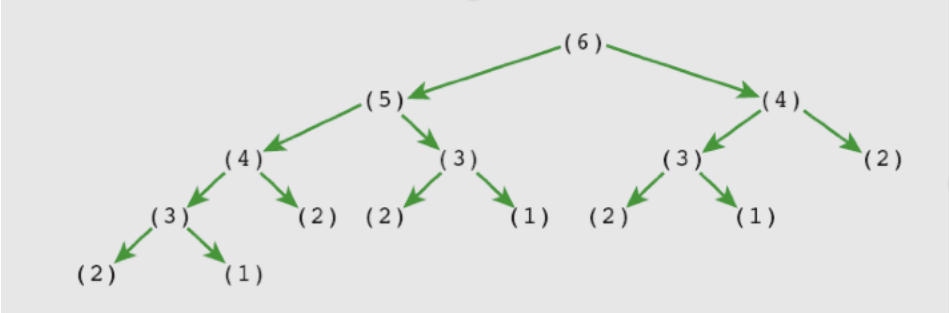

Figura 1.5 Esecuzione del programma ricorsivo Fibonacci per $n = 6$.

Osserviamo inoltre che le funzioni $F(2)$ e $F(1)$ sono attivate rispettivamente 5 e 3 volte e che il numeor complessivo di attivazione della funzione è pari a 14, maggiore del valore di $F(6)$ che è pari a 8.

La valutazione esatta del numero di attivazioni è più complessa dei casi precedenti. Nel seguito ci limitiamo a fornire una valutazione approssimata per eccesso e una per difetto che insime forniscono una valutazione coerente con la notazione O.

Osserviamo che $T(n)$ è definito dalla seguente equazione
$T(n) = 2$ se $n = 1, 2$ e $T(n) = T(n-1) + T(n-2) + 2$ se $n \geq 3$
osserviamo che in base alla definizione abbiamo che per ogni $n \geq 3$
$T(n-1) > T(n-2)$
e quindi possiamo scrivere
$T(n) = 2$ se $n = 1, 2$ e $T(n) \leq 2T(n-1) + 2$ se $n \geq 3$
srotolando l'equazione precedente otteniamo

$$\begin{aligned} T(n) &\leq 2 + 2T(n-1) \\ &\leq 2 + 2(2T(n-2) + 2) = (2+4) + 4T(n-2) \\ &\leq (2+4) + 4(2T(n-3) + 2) = (2+4+8) + 8T(n-3) \\ &\leq (2+4+8) + 8(2T(n-4) + 2) = (2+4+8+16) + 16T(n-4) \\ &\leq \dots \\ &\leq (2+4+8+16 \dots 2^k) + 2^k T(n-k) \end{aligned}$$

ovviamente ci fermiamo quando k è pari a n ottenendo
$T(n) \leq \sum_{k=1}^{n} 2^k = 2^{n+1} - 1$

Osservando che $2^{n+1} - 1 \leq 2(2^n)$ possiamo pertanto concludere che il costo dell'algoritmo è $O(2^n)$. Ricordiamo che la dimensioni dell'input è $O(\log n)$, pari al numero di bit sufficienti per memorizzare n e quindi concludiamo che il costo dell'algoritmo ricorsivo è esponenziale nel valore del parametro n e doppiamente esponenziale nelle dimensioni dell'input.

Un'analisi dettagliata, che va oltre lo scopo di queste dispense, dimostra che $T(n) = O(\Phi^n)$ dove $\Phi = 1 + sqrt(5))/2 \approx 1,6180339887$ è il numero che rappresenta la sezione aurea.

Consideriamo ora il seguente frammento di programma iterativo per il calcolo dei numeri di Fibonacci.

```
if(N==0 || N==1) fib = 1;
else{
     fib1=1;
     fib2=1;
     i=1;
     while(i<N){
             fib=fib1+fib2;
             i++;
             fib2=fib1;
             fib1=fib;
              }
       }
printf("Il valore di fibonacci di %d e': %d\n",N,fib)
```

In questo caso è facile verificare che il costo è $O(k)$ dove k è il numero di iterazioni del ciclo. È facile vedere che questo numero è pari a n. Quindi possiamo concludere che il costo del secondo programma è $O(n)$, che ricordiamo è esponenziale nella dimensione dell'input.

L'importanza dell'esempio fatto evidenzia che la complessità di programmi ricorsivi può essere molto maggiore del corrispondente progemma iterativo. Infatti può accadere molto spesso che versioni ricorsive abbiano tempi di esecuzione molto maggiori delle corrispondenti versioni itertive.

1.5 Delimitazione superiore e inferiore alla complessità di problemi

I concetti delimitazione superiore e inferiore alla complessità (upper e lower bound), usati per caratterizzare i costi asintotici di caso peggiore degli algoritmi, possono essere definiti anche in relazione ai problemi, piuttosto che agli algoritmi che risolvono problemi.

Diremo che un problema $\mathcal{P}$ ha una delimitazione superiore (*upper bound*) $O(f(n))$ se esiste un algoritmo $A_{\mathcal{P}}$ che risolve $\mathcal{P}$, con $C_{A_{\mathcal{P}}}(n) \in O(f(n))$. In altre parole, un algoritmo che risolve un determinato problema gli "trasferisce" l'upper bound. Ad esempio, poiché `ricercaSequenziale` ha upper bound $O(n)$, potremo dire che il problema della ricerca in una tabella ordinata ha upper bound lineare; tuttavia lo stesso problema è risolto anche da `ricercaBinaria`, che ha upper bound $O(\log n)$, il che ci porta a dire anche che la ricerca su tabella ordinata ha upper bound logaritmico. Fra tutti gli upper bounds applicabili a un dato problema è naturalmente più interessante quello asintoticamente inferiore, mentre gli altri upper bounds vengono considerati banali. Di fatto, parlando dell'upper bound di un problema, si fa implicitamente riferimento al miglior algoritmo noto, quello con il costo asintotico minore.

La definizione di lower bound introduce, rispetto al caso precedente, alcune peculiarità. Un problema $\mathcal{P}$ ha *lower bound* $\Omega(f(n))$ se per ogni algoritmo $A_{\mathcal{P}}$ che risolve $\mathcal{P}$ risulta $C_{A_{\mathcal{P}}}(n) \in \Omega(f(n))$. In questa definizione, a differenza della precedente, è impiegato il quantificatore universale piuttosto che quello esistenziale e, come conseguenza, viene a mancare una procedura costruttiva per determinare la delimitazione inferiore. Ad esempio, con riferimento al problema dell'ordinamento di un insieme di valori sul quale è definita una relazione d'ordine totale, è noto che l'algoritmo

Selection Sort ha costo $\Omega(n^2)$; ciò non significa che non esistono altri algoritmi con lower bound inferiore: infatti il Mergesort che abbiamo considerato ha costo $\Omega(n \log n)$.

La mancanza di una procedura costruttiva per il calcolo del lower bound di un problema ne rende più difficoltosa l'identificazione. Lower bounds sono di norma provati con tecniche più complesse basate su alberi di decisione, riduzioni fra problemi, circuiti algebrici ecc. In molti casi vale il cosiddetto lower bound banale, in cui si tiene conto che qualunque sia l'algoritmo risolvente, esso non può non leggere l'intero input. In questi casi si ottiene il lower bound banale di $\Omega(n)$.

Quando accade che un problema ammette la stessa funzione $f(n)$ come upper e lower bound diciamo che $f(n)$ è la *complessità intrinseca* del problema e tutti gli algoritmi che lo risolvono con costo $O(f(n))$ vengono qualificati come *ottimali*. Un naturale obiettivo per ogni progettista di algoritmi è progettare algoritmi ottimali.

Vediamo ora due esempi di delimitazione inferiori alla complessità di problemi. Considera il problema di ordinare n numeri distinti. Qualsiasi algoritmo di ordinamento deve essere in grado di distinguere tra il n! permutazioni di questi n numeri, dal momento che deve trattare ogni permutazione in modo diverso per poterla ordinare.

Complessità inferiore dell'Ordinamento

Come l'esempio precedente ci imitiamo a considerare algoritmi che utilizzano confronti fra dati per stabilire l'ordinamento come sono ad esempio gli algoritmi di ordinamento a bolle e di ordinamento per fusione precedentemente considerati. Alla fine della sezione vedremo che esistono algoritmi di ordinamento di tipo diverso.

Se limitiamo la nostra attenzione ad algoritmi di ordinamento basati su decisioni osserviamo che ciascuna decisione (o test) fornisce un singolo bit di informazione (ovvero se i due numeri confrontati sono in ordine o meno). Inoltre osserviamo che un algoritmo di ordinamento]uò essere visto come un metodo che dato una lista L di n numeri deve trovare la permutazione p che applicata a L fornisce una lista ordinata.

Ad esempio se $L = \{5, 1, 10, 3\}$ la permutazione p che cerchiamo è $p = (3, 1, 4, 2)$ che specifica come 5 (il primo numero di L) debba andare in terza posizione, 1 (il secondo numero di L) debba andare in prima posizione e così via.

La discussione precedente evidenzia come i possibili ordinamenti di n interi siano $n!$ (n fattoriale) tanti quanti le possibili permutazioni di n elementi. Pertanto se utlizziamo algoritmi di ordinamento basati su decisioni, il numero minimo di test da effetture deve permettere di distinguere la permutazione che ordina fra le $n!$ possibili permutazioni.

Dato che ciascuna decisione fornisce un singolo bit ne consegue che il numero di test da necessario per distinguere tra il $n!$ le permutazioni sono $\log_2(n!)$; è un limite inferiore per la complessità di qualsiasi algoritmo di ordinamento. Questo limite inferiore è noto come limite inferiore teorico dell'informazione.

Dimostriamo ora che $\log_2(n!) = \Omega(n \log_2 n)$.

Supponiamo n sia pari. In qusto caso è facile verificare la seguente disuguaglianza

$$n! = n \cdot (n-1) \cdot (n-2) \ldots 3 \cdot 2 \geq (n/2)^{n/2}$$

Perciò,

$$\log_2(n!) \geq \log_2(n/2)^{n/2} = (n/2) \cdot \log_2(n/2) = (n/2) \cdot ((\log_2 n) - 1) = \Omega(n \log n)$$

Se n è dispari osservaimo che $n/2 > \lfloor n/2 \rfloor$ e quindi possiamo ottenere la stessa valutazione asintotica nel seguente modo.

$$n! = n \cdot (n-1) \cdot (n-2) \ldots 3 \cdot 2 \geq \lfloor n/2 \rfloor^{\lfloor n/2 \rfloor + 1}$$

Perciò,

$$\log_2(n!) \geq \log_2 \lfloor n/2 \rfloor^{\lfloor n/2 \rfloor} \geq (n-1)/2 \cdot \log_2(n/2) = (n-1)/2 \cdot ((\log_2 n) - 1) = \Omega(n \log n)$$

Quindi ha bisogno di almeno log (n!) Elemento Ω (n log (n)) confronti per distinguere tra il n! possibili permutazioni di n numeri distinti. Ciò significa che Ω (n log (n)) è un limite inferiore per la complessità temporale di qualsiasi algoritmo di ordinamento basato su confronti.

Gli algoritmi di ordinamento Heapsort und Mergesort hanno un limite superiore di O (n log (n)). Pertanto, sono ottimali dal momento che raggiungono il limite inferiore.

Esistono algoritmi di ordinamento che non sono basati su confronti, ma utilizzano informazioni sui dati.

Ad esempio supponiamo di sapere che la sequenza da ordinare è composta solo da 1 e 2. In questo caso se dobbiamo ordinare il vettore

$$1, 2, 2, 1, 1, 2, 2, 1, 1, 1$$

possiamo utilizzare un metodo più veloce che sfrutta questa conoscenza. Infatti in questo caso basta contare quanti sono gli 1 e quanti i 2; in questo caso con una scansione troviamo che la sequenza da ordinare è composta da sei 1 e quattro 2 e quindi possiamo dedurre che la lista ordinata è

$$1, 1, 1, 1, 1, 1, 2, 2, 2, 2$$

Esistono algoritmi di ordinamento che sfruttano in modo più generale qusto approccio come ad esesmpio i metodi noti come Bucket sort (ordinamento a secchio) e Radix sort che se i dati di ingresso verifcano opportune ipotesi hanno complessità temporale pari a $O(n)$. Tuttavia, questi algoritmi non sono tanto generali quanto gli algoritmi basati sul confronto poiché si basano su determinati presupposti relativi ai dati da ordinare.

1.6 Conclusioni

In questo capitolo abbiamo visto come si valuta il costo di un programma e di un algoritmo. A questo scopo abbiamo introdotto diverse ipotesi semplificative che facilitano l'analisi della complessità di un programma, ma che la rendono approssimata. Abbiamo poi considerato diversi algoritmi. In conclusione valutiamo i criteri in base ai quali conviene scegliere l'algoritmo di soluzione. Infatti, quando dobbiamo risolvere un problema possiamo generalmente scegliere tra diversi algoritmi di soluzione e non sempre è chiaro quale sia la scelta migliore. Le proprietà che desideriamo dal nostro algoritmo sono principalmente due:

1. l'algoritmo deve essere semplice, in modo tale da facilitarne la comprensione, la programmazione e la correzione del programma;
2. l'algoritmo deve avere una complessità il più bassa possibile.

La prima proprietà fa riferimento al costo umano necessario per la scrittura del programma, mentre la seconda fa riferimento al costo di esecuzione del programma. Le due proprietà sono spesso in contraddizione fra loro. Regole precise che aiutino a decidere quale algoritmo applicare nei diversi casi non esistono. L'unico criterio che possiamo dare è il seguente:

- quando il programma deve essere eseguito poche volte su input aventi piccole dimensioni, allora è generalmente più importante cercare la semplicità del programma, perché, in questo caso il costo umano prevale sul costo della macchina;
- quando, invece, il programma deve essere eseguito molte volte su input aventi dimensioni grandi, allora conviene cercare di ottenere algoritmi efficienti perché il risparmio di risorse della macchina può convenire rispetto al maggior tempo che si richiede per la progettazione e la scrittura del programma.

Infine osserviamo che alcuni degli algoritmi considerati risultano intrinsecamente ricorsivi. In questi casi si possono scrivere procedure ricorsive molto semplici. Osserviamo però che programmi ricorsivi sono nella pratica inefficienti. Infatti ogni volta che viene chiamata una procedura,

l'elaboratore deve eseguire numerose operazioni (la creazione di una zona di memoria libera per le variabili della procedura, il passaggio dei parametri, ecc.), le quali richiedono un certo tempo di esecuzione aggiuntivo.

Il metodo di valutazione del costo del programma che abbiamo introdotto suppone che tutte le operazioni relative alla chiamata di una procedura abbiano costo unitario. La semplificazione introdotta non crea problemi se il numero delle chiamate di procedura è piccolo. Questo non è più vero nel caso di programmi ricorsivi. Ad esempio, per ordinare un vettore di 128 elementi con l'algoritmo per fusione, sono necessarie molte chiamate ricorsive. Questo significa che le costanti moltiplicative "nascoste" dalla notazione O sono molto grandi. Quindi su insiemi piccoli algoritmi come l'ordinamento a bolle risultano più veloci. Solo su insiemi molto grandi si riesce a percepire la differenza, come abbiamo visto nell'esercizio **??**.

Tuttavia, è utile ricordare che la ricorsione è un metodo che permette di scrivere programmi più semplici, ma non è indispensabile. Infatti, dato un qualunque programma ricorsivo è sempre possibile ottenere un programma equivalente non ricorsivo che solitamente è più efficiente (poiché ha costanti moltiplicative più basse).

1.7 Esercizi

Esercizio 1. Quali delle seguenti affermazioni sono vere?

1. Un algoritmo $\Theta(n)$ è $O(n)$
2. Un algoritmo $\Theta(n)$ è $O(n^2)$
3. Un algoritmo $\Theta(n^2)$ è $O(n^3)$
4. Un algoritmo $\Theta(n)$ è $O(1)$
5. Un algoritmo $O(1)$ è $\Theta(1)$
6. Un algoritmo $O(n)$ è $\Theta(1)$

Esercizio 2. Si consideri la seguente definizione ricorsiva per il calcolo del fattoriale di un numero. Analizzare la complessità dell'algoritmo in funzione delle dimensioni dell'input.

```
int find_factorial(int n)
{
   //Factorial of 0 is 1
   if(n==0)
      return(1);

   //Function calling itself: recursion
   return(n*find_factorial(n-1));
}
```

Esercizio 3. Provare che il problema della ricerca in un insieme ordinato di n elementi ha costo $\Omega(\log n)$ se si utilizzano confronti. (Sugg. utilizzare un ragionamento analogo a quanto fatto per l'ordinamento. Rappresentare il problema come un albero la cui profondità rappresenta il costo nel caso peggiore di un algoritmo basato su confronti. Valutare la profondità dell'albero in funzione di n.

Capitolo 2

Problemi trattabili e problemi intrattabili

2.1 Problemi trattabili e problemi intrattabili

Abbiamo visto nel capitolo 2 che esistono problemi che non possono essere risolti in modo automatico, ovvero linguaggi indecidibili, ci chiediamo in questo capitolo quanto costi invece risolvere quelli decidibili. In particolare, in questa parte delle dispense ci concentreremo sul tempo necessario affinchè una macchina di Turing possa decidere se un determinato input appartiene o meno a uno specifico linguaggio. In base a tale misura di complessità, definiremo la classe P, ovvero la classe dei problemi risolubili efficientemente, e mostreremo alcuni esempi di linguaggi in tale classe, facendo anche uso del concetto di riducibilità polinomiale.

2.1.1 La classe P

Nel capitolo precedente abbiamo introdotto l'analisi asintotica per facilitare il calcolo della complessità temporale di un programma utilizzando la notazione O, che consente di ignorare fattori moltiplicativi costanti e termini di ordine inferiore al crescere delle dimensioni dell'input.

Ricordiamo inoltre che la funzione che descrive la complessità di calcolo è una funzione nelle dimensioni dell'input. Se l'input contiene numeri assumiamo che questi siano rappresentati in base 2. Osserviamo che questa rappresentazione dei numeri implica che il numero di bit utilizzato è al massimo una unità più grande del logaritmo del numero; ad esempio rappresentiamo $1024 = 2^{10}$ con undici bit (10000000000). In questo modo escludiamo per i numeri rappresentazioni che utilizzano un numero di bit pari al valore del numero (ad esempio come rappresentare cinque con una sequenza di cinque uni, 11111).

L'uso di tali codifiche standard ci permetterà di assumere che tutte le rappresentazioni ragionevoli di un problema siano tra di loro scambiabili, ovvero, data una di esse, sia possibile passare a una qualunque altra rappresentazione in un tempo polinomiale rispetto alla rappresentazione originale. Tali considerazioni ci portano a fornire la seguente definizione.

Definizione 3. La classe P. La classe P è l'insieme dei linguaggi L per i quali esiste un programma Python che decide L e per cui il tempo di esecuzione su input di dimensione n è $O(n^k)$ per qualche $k \geq 0$.

Osserviamo che nella definizione precedente la restrizione relativa ad un programma Python non é restrittiva. Infatti se invece di un programma Python usiamo un altro linguaggio di programmazione o anche una macchina di Turing otteniamo la stessa definizione. Possiamo quindi dare anche la seguente definizione equivalente.

Definizione 4. La classe P. La classe P è l'insieme dei linguaggi L per i quali esiste una macchina di Turing con un solo nastro che decide L e per cui il numero di passi di calcolo su input di dimensione n é $O(n^k)$ per qualche $k \geq 0$.

Le due definizioni precedenti sono equivalenti. Infatti, è possibile dimostrare che se esiste una macchina di Turing che risolve un problema (o riconosce un linguaggio) in tempo polinomiale nella lunghezza dell'input se e solo se esiste un programma in un linguaggio di programmazione che risolve il problema e richiede tempo polinomiale per la sue esecuzione. Non dimostreremo questo teorema che richiede di mostrare due cose: innanzitutto che per ciascuna istruzione Python esiste una macchina di Turing che esegue l'istruzione in tempo polinomiale; inoltre è necessario mostrare come combinare macchine di Turing che eseguono singole istruzioni in un unica macchina (così come combiniamo singole istruzioni in un unico programma).

La definizione della classe P risulta, quindi, essere robusta rispetto sia a modifiche del modello di calcolo utilizzato che rispetto a modifiche della codifica utilizzata, purché questi siano entrambi ragionevoli. Tuttavia l'importanza di tale classe risiede nel fatto che essa viene comunemente identificata con l'insieme dei linguaggi trattabili, ovvero risolubili in tempi ragionevoli.

Per convincerci di tale affermazione, consideriamo nuovamente la tabella 3. È evidente dalla tabella che i primi due algoritmi hanno una complessità decisamente ragionevole, che il terzo algoritmo sia poco utilizzabile per istanze di lunghezza superiore a 1000 e che il quarto algoritmo (quello con complessità 2^n) risulta del tutto inutilizzabile per istanze di lunghezza superiore a 40. Infatti per risolvere un problema di dimensione 100 con un algoritmo di complessità 2^n su un elaboratore in grado di eseguire un milione di operazioni al secondo sono necessari più di 32 milioni di milioni di milioni di anni!

Scegliendo una complessità temporale più favorevole di quello di complessità 2^n ma non polinomiale (ad esempio, una complessità sub-esponenziale del tipo $O(k \log h(n))$ con k e h costanti), le differenze sarebbero meno significative per piccoli valori di n, ma riapparirebbero comunque al crescere di n. Possiamo, quindi, concludere che se il nostro obiettivo è quello di usare il calcolatore per risolvere problemi la cui descrizione includa un numero relativamente alto di elementi, allora è necessario che l'algoritmo di risoluzione abbia una complessità temporale polinomiale: per questo motivo, la classe P è generalmente vista come l'insieme dei problemi risolvibili in modo efficiente.

Esempi di linguaggi in P

Data l'equivalenza tra le macchine di Turing e i linguaggi di programmazione di alto livello, che abbiamo stabilito nel quarto capitolo, nel resto di questo capitolo ci limiteremo a descrivere gli algoritmi facendo riferimento a questi ultimi: in realtà, possiamo essere ancora più pigri e fare uso del linguaggio naturale, lasciando al lettore interessato il compito di tradurre tali descrizioni degli algoritmi in un programma scritto in un qualunque linguaggio di programmazione.

Ad esempio i programmi che abbiamo considerato nella sezione precedente per la ricerca di un elemento in un punto, il calcolo del valore di un polinomio in un punto, per il calcolo del massimo comun divisore di due numeri sono esempi di programmi in P.

2.1.2 La classe NP

Migliaia di problemi interessanti da un punto di vista applicativo appartengono alla classe P. Tuttavia, è anche vero che migliaia di altri problemi, altrettanto interessanti, sfuggono a una loro risoluzione efficiente. Molti di questi, però, hanno una caratteristica in comune che giustificherà l'introduzione di una nuova classe di linguaggi, più estesa della classe P: per capire tale caratteristica, consideriamo i seguenti quattro esempi di problemi computazionali. Il problema SAT, come detto, consiste nel decidere se una data formula booleana in forma normale congiuntiva è soddisfacibile. Non si conoscono algoritmi polinomiali che risolvono questo problema, ma possiamo osservare che se una formula F con n variabili e m clausole è soddisfacibile, allora esiste un'assegnazione che soddisfa F. Un'assegnazione altro non è che una stringa di n valori binari (che codificano i valori delle n variabili booleane); pertanto tale assegnazione può essere codificata

con una stringa di lunghezza polinomiale nella lunghezza della codifica di F. Inoltre, esiste un semplice algoritmo polinomiale che verifica se, effettivamente, un'assegnazione di valori di verità soddisfa la formula: tale algoritmo deve semplicemente sostituire alle variabili i valori assegnati e verificare che ogni clausola sia soddisfatta.

Ovvamente tutti i problemi che appartengono alla classe P appartengono anche a NP. Infatti una macchina di Turing deterministica è un caso speciale di macchina nondeterministica. Quindi possiamo dire che la classe P è inclusa nella classe NP. Formalmente abbiamo

$$P \subseteq NP$$

Non sappiamo se l'equivalenza precedente sia stretta o meno. Per mostrare che l'equivalenza è stretta (cioè che $P \subset NP$) dobbiamo mostrare che esiste un problema in NP che *non* appartiene a P. Allo stato attuale della conocenza pensiamo che questo sia vero ma non abbiamo ancora una prova. Infatti stabilire se $P = NP$ o se $P \subset NP$ è un problema aperto da molto tempo ed è anche noto come il problema da un milione di dollari (infatti questa è la cifra che chi risolve per primo il problema riceverà da una fondazione americana).

Il problema SAT, che richiede di riconoscere se una formula logica sia soddisfacibile è un esempio di problema in NP per cui non sappiamo se appartenga a P (infatti si crede che non appartenga a P).

La classe NP é molto ricca e contiene molti problemi che probabilmente non appartengono a P. Un elenco completo é impossibile. Tanto per dare un'idea del tipo di problemi alla classe ecco un breve elenco:

- il problema di partizionare un insieme in due parti uguali. In particolare, sono dati n numeri interi $a_1, a_2, \ldots, a_n$ e ci chiediamo se sia possibile dividerli in due sottoinsiemi disgiunti tale che la somma dei due sottoinsiemi sia uguale fra loro.

- il problema di colorare una mappa. Data una carta geografica che rappresenta nazioni, ci chiediamo se sia possibile colorare tutti gli stati della mappa con tre colori in modo tale che due stati confinanti abbiano colori diversi.

- il problema di trovare il percorso più breve per un insieme di punti. In questo caso sono dati n punti nel piano e un valore x e ci chiediamo se esiste un percorso che collega tutti i punti dati con segmenti rettilinei, abbia lunghezza minore di x e non passi due volte per lo stesso punto.

2.1.3 Oltre NP

In quest'ultimo paragrafo introdurremo due classi di complessità che includono la classe NP e che molto probabilmente sono strettamente pi'u grandi di quest'ultima. La prima classe è la classe EXP, che include tutti i problemi e i linguaggi decidibili in tempo esponenziale; la seconda classe è la classe $PSPACE$, che include tutti i linguaggi decidibili usando un numero polinomiale di celle di memoria.

Rimandiamo alle referenze per una trattazione formale delle proprietà di queste classi e ci limitiamo a stabilire le loro relazioni con P e NP.

In particolare abbiamo le seguenti relazioni fra le classi finora definite.

- $P \subseteq NP$. Come abbiamo discusso in precedenza un algortimo deterministico è un caso speciale di algortimo non deterministico.

- $NP \subseteq PSPACE$. La dimostrazione di questa relazione può essere dimostrata nel seguente modo. Un algoritmo non deterministico che ha tempo di esecuzione polinomiale può utilizzare al massimo uno spazio polinomiale (utilizza nel caso peggiore un numero di celle di memoria pari al numero di passi di calcolo); inoltre l'emulazione di un algoritmo nondeterministico che richiede tempo polinomiale può essere fatta usando uno spazio polinomiale che è sufficiente a simulare tutte le possibili scelte.

- $PSPACE \subseteq EXP$. La dimostrazione di questa relazione si basa sulla medesima intuizione del caso precedente. Infatti una macchina di Turing con n stati e che utilizza $s$celle del nastro può trovarsi al massimo in un numero di configurazioni diverse pari a $O(n2^s)$ (infatti se la macchina usa s celle del nastro le configurazioni diverse del nastro sono $O(2^s)$). Questo implica che se la macchina impiegasse più di $O(n2^s)$ passi, allora avremmo che necessariamente esistono due configurazioni del nastro identiche in cui la macchina si trova nello stesso stato. Questo implica che necessariamente la macchina non termina e cicla per sempre.

Le relazioni precedenti fra le classi di complessità sono così riassunte

$$P \subseteq NP \subseteq PSPACE \subseteq EXP$$

Non sappiamo se le relazioni fra le diverse classi siano o meno proprie; tuttavia abbiamo evidenza di problemi naturali che appartengono ad una classe ma che si crede non appartengono alla classe precedente. Ad esempio sappiamo che il problema di sapere se una formula booleana con quantificatori[1]

2.2 Conclusioni

In questo capitolo abbiamo visto come si valuta il costo di un programma e di un algoritmo. A questo scopo abbiamo introdotto diverse ipotesi semplificative che facilitano l'analisi della complessità di un programma, ma che la rendono approssimata. Abbiamo poi considerato diversi algoritmi. In conclusione valutiamo i criteri in base ai quali conviene scegliere l'algoritmo di soluzione. Infatti, quando dobbiamo risolvere un problema possiamo generalmente scegliere tra diversi algoritmi di soluzione e non sempre è chiaro quale sia la scelta migliore. Le proprietà che desideriamo dal nostro algoritmo sono principalmente due:

1. l'algoritmo deve essere semplice, in modo tale da facilitarne la comprensione, la programmazione e la correzione del programma;

2. l'algoritmo deve avere una complessità il più bassa possibile.

La prima proprietà fa riferimento al costo umano necessario per la scrittura del programma, mentre la seconda fa riferimento al costo di esecuzione del programma. Le due proprietà sono spesso in contraddizione fra loro. Regole precise che aiutino a decidere quale algoritmo applicare nei diversi casi non esistono. L'unico criterio che possiamo dare è il seguente:

- quando il programma deve essere eseguito poche volte su input aventi piccole dimensioni, allora è generalmente più importante cercare la semplicità del programma, perché, in questo caso il costo umano prevale sul costo della macchina;

- quando, invece, il programma deve essere eseguito molte volte su input aventi dimensioni grandi, allora conviene cercare di ottenere algoritmi efficienti perché il risparmio di risorse della macchina può convenire rispetto al maggior tempo che si richiede per la progettazione e la scrittura del programma.

Infine osserviamo che alcuni degli algoritmi considerati risultano intrinsecamente ricorsivi. In questi casi si possono scrivere procedure ricorsive molto semplici. Osserviamo però che programmi ricorsivi sono nella pratica inefficienti. Infatti ogni volta che viene chiamata una procedura, l'elaboratore deve eseguire numerose operazioni (la creazione di una zona di memoria libera per le variabili della procedura, il passaggio dei parametri, ecc.), le quali richiedono un certo tempo di esecuzione aggiuntivo.

Il metodo di valutazione del costo del programma che abbiamo introdotto suppone che tutte le operazioni relative alla chiamata di una procedura abbiano costo unitario. La semplificazione

[1] Una formula boooleana quantificata è una formula booleana in cui le variabili devono soddisfare ulteriori vincoli, che esprimono concetti del tipo *p*er ogni valore della variabile x oppure *e*siste un valore della vairabile x.

introdotta non crea problemi se il numero delle chiamate di procedura è piccolo. Questo non è più vero nel caso di programmi ricorsivi. Ad esempio, per ordinare un vettore di 128 elementi con l'algoritmo per fusione, sono necessarie molte chiamate ricorsive. Questo significa che le costanti moltiplicative "nascoste" dalla notazione O sono molto grandi. Quindi su insiemi piccoli algoritmi come l'ordinamento a bolle risultano più veloci. Solo su insiemi molto grandi si riesce a percepire la differenza, come abbiamo visto nell'esercizio **??**.

Tuttavia, è utile ricordare che la ricorsione è un metodo che permette di scrivere programmi più semplici, ma non è indispensabile. Infatti, dato un qualunque programma ricorsivo è sempre possibile ottenere un programma equivalente non ricorsivo che solitamente è più efficiente (poiché ha costanti moltiplicative più basse).