FABRIZIO LUCCIO

INFORMATICA PER LE BIOTECNOLOGIE

Prima Parte: Concetti generali

0 UN'INTRODUZIONE

Cari studenti, se per ragioni di necessità o gusto personale il vostro unico e rispettabilissimo scopo è superare l'esame, potete tranquillamente saltare questa introduzione. Ma poiché per la maggioranza di voi il periodo dello studio potrebbe concludersi all'università, e poiché gli algoritmi sono i principali attori di questo corso, vorrei parlarvi brevemente dell'origine stessa del termine, sconosciuto ai più fino a pochi anni fa e che ora circola ovunque non sempre a proposito: termine legato al nome di uno dei più grandi scienziati della storia, cresciuto in un luogo e in un ambiente magici e irripetibili. Dunque, se vi va, buona lettura anche dell'introduzione. Se qualcuno di voi volesse approfondire l'argomento, ma la cosa diviene parecchio impegnativa, il miglior riferimento, pur con qualche imperfezione matematica, è il voluminoso saggio storico di S.Frederick Starr pubblicato da Princeton University Press nel 2013, di cui esiste l'ottima (e costosa) traduzione in italiano: "L'illuminismo perduto. L'età dell'oro dell'Asia centrale dalla conquista araba a Tamerlano". A questo libro mi sono largamente ispirato per scrivere la breve nota seguente.

Dixit Algorithmi

Nei primi decenni del XII secolo il filosofo Adelardo di Bath lasciò la natia Inghilterra per lunghe peregrinazioni in cerca delle piú profonde fonti del sapere. In particolare era noto che gli scienziati e i filosofi arabi avevano raggiunto da secoli mete sconosciute all'Europa medievale costruendo le loro opere su quelle della Grecia classica che erano state in gran numero tradotte in arabo[1]. Attraversate la Francia e l'Italia Adelardo raggiunse le coste del Mediterraneo orientale e tornò in patria con molte copie di testi in arabo, alcuni dei quali erano stati scritti duecento anni prima da uno dei più grandi uomini di scienza che il mondo abbia conosciuto: **Abū Ja'far Muhammad Ibn Mūsā al-Khwārizmī,** letteralmente Muhammad, padre (Abū) di Ja'far, figlio (Ibn) di Mosè, proveniente da Khwārizm. Brevemente citato come **al-Khwārizmī**, secondo la *nisba*, cioè provenienza, con cui nell'uso arabo ancor oggi si indicano spesso le persone.

Tra le opere di al-Khwārizmī di cui Adelardo curò la traduzione in latino ve n'è una senza titolo in cui si descrive il sistema di rappresentazione indiano dei numeri in notazione posizionale con lo zero, che per questa via fu introdotta in Europa con il nome improprio di numerazione araba; e si sostiene l'importanza di quella notazione indicando come utilizzarla nelle operazioni aritmetiche al posto dei numeri romani inefficentissimi a tale scopo. Adelardo apre la traduzione con le parole: *Dixit Algorithmi*, asserendo nella latinizzazione del nome dell'autore la assoluta autorità di questo. Da allora il termine *algorithmus* entrò (molto lentamente) nell'uso in Europa per indicare via via non più l'autore ma i procedimenti di calcolo da lui descritti.

[1]Dall'VIII secolo, prima i persiani poi gli arabi avevano istituito grandi centri di traduzione di opere classiche, in particolare greche e indiane: prime tra tutte quelle di Aristotele considerato il "primo maestro". Solo attraverso le traduzioni in arabo molti classici greci sono poi tornati in Europa ove gli originali erano perduti.

Secondo la nisba lo scienziato proveniva dalla regione di Khwārizm in Asia centrale [2], zona che nella geografia politica di oggi si estende tra l'Iran nord-orientale, l'Afghanistan nord-occidentale, e il Turkmenistan meridionale, e che vantava importantissime città come Nishapur, Balkh e Merv tra tante altre che oggi sono note solo all'archeologia (vedi mappa nella figura 0 in fondo al capitolo). Qui sarebbero fiorite con straordinario fervore le scienze principali del tempo inquadrate nella "filosofia" (*al-falsafa*), in particolare la matematica, l'astronomia e la medicina, ma anche l'antropologia, la sociologia, la psicologia, la politica e la musica, dando i natali a una numerosissima schiera di grandi studiosi.

Tutti gli adepti di al-falsafa difendevano la forza della ragione dai possibili attacchi di tutte le religioni conviventi nell'area, cercando di dimostrare, forse con più opportunismo che sincerità, che ragione e dogmi potevano convivere perché si rivolgevano a diversi livelli della conoscenza; finché poco a poco l'intransigente Islam radicale spense un movimento culturale paragonabile ai più importanti eventi della storia dell'uomo. La regione era inclusa nel Khorasan, in persiano la "Terra dove nasce il sole". Una terra assai più vasta, che nel corso di quattro secoli si sarebbe largamente estesa verso nord-est fino a includere praticamente tutta l'Asia centrale, dalla Persia al confine con la Cina, divenendo nei fatti uno stato sovrano che manteneva solo un'incerta dipendenza formale dal califfato arabo.

Se le opere di al-Khwārizmī sono notissime, poco si sa della sua vita. Nacque nel 780 nella Corasmia, la distesa di deserti a sud-est del lago d'Aral. La sua lingua d'origine era una forma di antico persiano. Studiò a Merv, importantissima città turcofona del Khorasan collocata nel Turkmenistan di oggi e principale centro culturale dell'Asia di quel tempo, precedentemente distrutta dai conquistatori arabi ma rapidamente rifiorita sotto gli Abbasidi tanto che il Califfo al-Ma'mun vi aveva stabilito la sua sede. Presumibilmente a Merv al-Khwārizmī apprese l'arabo, lingua in cui avrebbe redatto tutte le sue opere. Verso i quarant'anni, perfettamente formato come matematico e astronomo, si trasferì a Baghdad dove al-Ma'mun era tornato fondando e frequentando egli stesso una "Casa della Sapienza" in cui gli studiosi potevano perseguire le loro ricerche senza alcun condizionamento, sostenuti anche economicamente dallo stato. Una straordinaria accademia frequentata in prevalenza da studiosi del Khorasan, ma anche da arabi, da ebrei e cristiani provenienti da ovest e buddisti provenienti da est; e che due secoli più tardi sarebbe caduta sotto i colpi dell'Islam sunnita che anteponeva i dogmi alla libera ragione. Nell' 850 al-Khwārizmī morì a Baghdad, 1500 km a ovest del Khorasan; e anche su questa città è opportuno fornire qualche notizia.

Fondata con il nome di Medinat al-Salam (Città della Pace) nel 762 sul Tigri, nell'Iraq di oggi, Baghdad vide la luce per ordine del califfo di allora al-Mansūr, in una landa desolata indicata da alcuni astrologi venuti da Merv. Benché in territorio arabo, anche gli architetti e i tecnici che progettarono la città e diressero i lavori, e in genere le persone di cultura, venivano dal lontanissimo Khorasan. E tra esse ebbero massima importanza i membri della famiglia Barmak di Balkh, nell'Afghanistan

[2] Il nome della regione si trova formulato in modi differenti nel nostro alfabeto a seconda della lingua in cui viene pronunciato, tra le tante parlate in quell'area.

di oggi: buddisti convertiti all'Islam divennero i principali consiglieri del califfo e in pochi anni dettero origine allo studio delle scienze, all'industria libraria e alla traduzione in arabo dei classici greci e indiani. Alla morte di al-Mansūr nel 775 divenne califfo il nipote ventenne Hārūn al-Rashid, ricordato nelle *Mille e una notte* assieme al Visir Ja'far, che dette ricchezza e lustro incomparabili al mondo arabo. Alla morte di Hārūn il califfato fu diviso tra i suoi due figli al-Amim (a Baghdad) e al-Ma'mūn (a Merv, come già detto) che presero immediatamente a fronteggiarsi con le armi finché al-Ma'mūn, appoggiato dalle fortissime milizie turcofone, ebbe il sopravvento uccidendo il fratello e riunificando il califfato a Baghdad ove si insediò nell' 819. E qui giunse anche al-Khwārizmī.

In brevissimo tempo dopo la fondazione, e soprattutto dopo il ritorno di al-Ma'mūn, Baghdad divenne la città più popolosa del mondo, con un travolgente fiorire di attività culturali che per volume e probabilmente qualità non ha avuto uguali nella storia dell'uomo. E tra le innumerevoli opere scientifiche ivi pubblicate, quelle di al-Khwārizmī spiccano per importanza sia teorica che pratica.

Con sorprendente semplicità al-Khwārizmī introduce meccanismi matematici nuovi attraverso una serie di problemi di natura varia, dall'ingegneria edile alla vita sociale, dalle regole giuridiche alla costruzione di congegni meccanici. Il suo famoso testo di matematica, che prese in occidente il titolo di *Algebra* in assonanza al suo metodo *al-jabr* (completamento) per la riduzione dei termini delle equazioni, descrive, rinnova e amplia in modo fondamentale i metodi di soluzione delle equazioni di primo grado e alcune del secondo intrapresi ai tempi dei babilonesi, costruendone una vera e propria teoria basata sui radicali. Un suo monumentale testo di *Tavole Astronomiche* esamina con estrema precisione il moto del sole, della luna e dei pianeti conosciuti arrivando a ipotizzare che i pianeti si muovano su orbite ellittiche e non circolari. Il *Libro sulla forma della Terra* che amplia opere greche già note descrivendo la geografia del mondo con la ridefinizione di terre emerse e oceani e la locazione precisa di luoghi e città, divenne anche in Europa il riferimento di base di ogni studio geografico.

Non è questa l'occasione per descrivere altre opere di al-Khwārizmī e di tanti importanti studiosi di quel tempo e luogo, ma mi sembra obbligatorio citarne brevemente almeno altri quattro che nel corso di quattro straordinari secoli, dal 750 al 1150 circa, influenzarono in modo determinante i successivi studi in Europa stabilendo un autentico ponte tra la cultura classica e il pensiero moderno. Sono, in ordine di tempo:

Ahmad ibn Muhammad ibn Muhammad al-Farghānī (797-870); brevemente **al-Farghānī**, o Alfraganus in Europa, per l'astronomia;

Abū Nasr Muhammad ibn Muhammad al-Farābī (870-950); brevemente **al-Farābī**, o Alfarabius in Europa, per la politica e la musica;

Abū 'Ali al-Husayn ibn Sīnā (980-1037); brevemente **ibn Sīnā**, o Avicenna in Europa, per la medicina;

Abū 'l-Fath Omar ibn Ibrahīm al-Kayyām Nīshāpūrī; brevemente **Omar Khayyām** (1048-1131), per la poesia e la matematica.

Al-Farghānī, giunto a Bhagdad da Merv al seguito del califfo al-Mamūn, si distinse per accurati calcoli di astronomia pubblicando un compendio dell'*Almagesto* di Tolomeo aggiornato con le sue ultime scoperte. Il trattato, divenuto famoso in Europa, fu il principale riferimento per Dante sulle nozioni astronomiche nella Divina Commedia e nel Convivio, e fu utilizzato due secoli dopo da Cristoforo Colombo per convincere la corte spagnola sulla correttezza della sua proposta di viaggio.

Al-Farābī si formò a Bukhara nell'Uzbekistan, città dove l'Islam sciita sempre in fluida attesa dell'Imam consentiva maggiore libertà di pensiero. Nella maturità si trasferì a Baghdad. Le sue considerazioni sull'ambito dell'indagine razionale separata dalla nozione di Dio come Primo motore immobile furono riprese dall'ebreo Maimonide e dal cristiano Tommaso d'Aquino, da Dante e da Kant. Pensatore inflessibile aveva concepito un trattato sul governo delle città ispirato al pensiero platonico, che costituiva un'implicita e coraggiosa critica del califfato; e per pubblicarlo si trasferì definitivamente a Damasco sotto la protezione dei Fatimidi sciiti. Come curiosità ricordo che, virtuoso suonatore di liuto, scrisse un *Grande libro della musica* sulla teoria musicale e la sua relazione con la matematica la cui traduzione divenne base della musicologia europea. Nonostante le sue idee antidogmatiche in urto con l'Islam sunnita, e benché provenisse dal Khorasan, al-Farābī è considerato dalla maggioranza degli arabi il loro massimo filosofo.

Ibn Sīnā, noto in tutta Europa come Avicenna, era cresciuto anche lui a Bukhara e iniziò sin da ragazzo a emergere come brillantissimo polemista in molti campi del sapere. Tra le sue opere il *Canone della medicina* tradotto in latino ebbe larghissima circolazione in Europa nei secoli seguenti dando i natali a una medicina autenticamente scientifica, e fu adottato in India come base di una scuola medica intestata al suo nome. Oltre a descrivere i trattamenti pratici per centinaia di malattie, il trattato emergeva come assoluta novità per l'esame degli effetti dell'ambiente sulla salute e per il riconoscimento dei disturbi psicosomatici come attinenti alla medicina, da cui forse si potrebbero ancora trarre utili consigli.

Omar Khayyām è conosciuto in Europa specialmente come poeta per le sue bellissime *quartine* apertamente critiche sull'incoerenza della religione dogmatica, ispirate ai piaceri della vita, alla meditazione sulla morte e ai limiti della ragione. Ma ha lasciato uno stupefacente trattato di matematica dove affronta per la prima volta la teoria delle equazioni di terzo grado e la possibilità di concepire una geometria non euclidea, anticipando concetti che l'Europa avrebbe "scoperto" molti secoli dopo.

È opportuno sottolineare che tutti questi studiosi provenivano dal Khorasan mentre moltissimi libri di storia e di scienze parlano delle loro opere come frutto della cultura araba. Gli studiosi del Khorasan *scrivevano* in arabo che era a quei tempi la lingua colta in tutti i paesi che gli arabi avevano conquistato, come avveniva con il latino in Europa e con l'inglese oggi, ma provenivano da una zona molto circoscritta del mondo ove la cultura era prevalentemente protopersiana e prototurca: con le parole del saggio di Fredrick Starr, considerare arabi questi studiosi è come considerare britannico uno scienziato giapponese che scrive in inglese i suoi articoli.

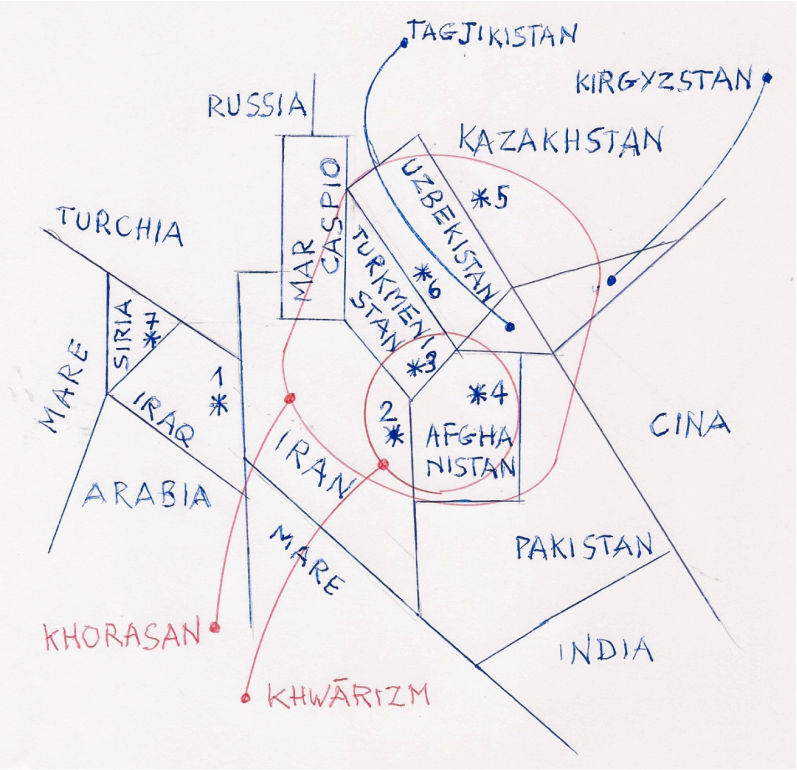


Figura 0. Asia centrale poligonale. Geografia politica attuale con indicazione delle città medievali: 1.Baghdad, 2.Nishapur, 3.Merv, 4.Balkh, 5.Farāb, 6.Bukhara, 7.Damasco.

al-Shayyan, l'incognita

Una caratteristica unica del trattato sull'algebra di al-Khwārizmī è il linguaggio praticamente privo di formule, tanto che le equazioni, pur risultando chiarissime da comprendere ed elaborare, vi sono descritte solo a parole. E tra queste ha un ruolo di base la parola *al-Shayyan*, letteralmente "qualcosa", o "l'incognita" nel nostro linguaggio matematico. Ed ecco una curiosià che non troverete nei libri.

Nella matematica di tutto il mondo, oggi anche in Cina e nei paesi arabi, l'incognita di un'equazione si indica con la lettera x. Nelle formule matematiche arabe, rielaborate principalmente in Spagna nell'XI secolo, la lettera era la Shīn con cui inizia la parola Shayyan. Nella pronunzia italiana la Shīn ha il suono sc di scimmia, ma non ha suono corrispondente in spagnolo. Per questo in Spagna fu scelta la lettera greca χ (chi), che prese poi la forma della x, graficamente molto simile e presente nella maggioranza degli alfabeti impiegati in Europa. E da allora l'incognita è x per tutti i matematici.[3]

[3]In alcuni libri di matematica, e sistematicamente su Web, si trova scritto che la scelta della x per indicare l'incognita si deve a Cartesio: che però visse nel 1600 . . . .

1 DUE ESEMPI PER COMINCIARE

Entriamo nel mondo degli algoritmi attraverso due problemi molto diversi tra loro, per richiamare concetti che incontreremo durante tutto il corso. Il primo richiede calcoli tra numeri e il secondo confronti tra oggetti, anche se in entrambi i casi sarebbe più esatto parlare di operazioni su sequenze di caratteri che rappresentano i dati.

La moltiplicazioe di interi

La computazione numerica esiste da tempi antichissimi, e il primo esempio per nulla banale appare in uno straordinario papiro egiziano del 1650 a.C circa, in cui è esposto un metodo per eseguire la moltiplicazione tra due numeri interi (gli egiziani, come i computer, conoscevano solo gli interi).[4] L'algoritmo, apparentemente molto diverso da quello che usiamo noi, è basato su addizioni, raddoppi e dimezzamenti presumibilmente a casusa dell'impiego di abaci che permettevano facilmente di eseguire queste operazioni elementari. Prima di esporlo giustifichiamo la sua correttezza spiegando i principi su cui si basa.

Indicando con A, B i fattori da moltiplicare e con P il loro prodotto:

1. se A è pari, $P = (A/2) \cdot (2B)$;
2. se a è dispari e maggiore di 1, $P = ((A-1)/2) \cdot (2B) + B$;

quindi il valore di P si può ottenere iterando i passi 1 e 2 finché il valore di A, che diminuisce a ogni passo, diviene 0.

Il metodo di Ahmes si chiama oggi **algoritmo** e la sua realizzazione in un linguaggio di programmazione per un calcolatore si chiama **coding**. Vediamo come questo appare in uno spezzone di "pseudo programma" basato sullo stile e sul tipo di linguaggio che impiegheremo nel seguito, confidando che se ne capisca senza problemi il significato.

input A, B; **output** P;
$P = 0$;
while $A \geq 1$
  { **if** (A è dispari) $P = P + B$;
    $A = \lfloor A/2 \rfloor$;
    $B = B \times 2$ };

[4] Il papiro, noto nel mondo scientifico come papiro di Ahmes dal nome dello scriba e nel mondo museale come papiro di Rhind dal nome del gentiluomo che se l'è portato via, è conservato nel British Museum anziché nel Museo del Cairo come dovrebbe, e se ne può ammirare solo una copia.

ove il simbolo = indica l'assegnazione del valore che appare alla sua destra, alla variabile a sinistra; i simboli $\lfloor x \rfloor$ indicano il valere di x approssimato all'intero immediatamente inferiore; e il comando **while** indica che le tre righe seguenti, racchiuse ta parentesi graffe, devono essere eseguite finché la condizione $A \geq 1$ non è più verificata perché A diviene uguale a zero.

Seguuiamo le operazioni di questo programma su un esempio: la moltiplicazione tra 37 e 12. La tabella seguente, percorsa da sinistra verso destra, mostra i valori che assumono via via le tre variabili P, A, B. Il risultato è $37 \times 12 = 444$.

P	0	12		60			444
A	37	18	9	4	2	1	0
B	12	24	48	96	192	384	768

Anticipiamo ora una domanda che ci porremo per tutti gli algoritmi. A parte l'irrinunciabile correttezza del risultato, è questo un buon algoritmo? come si confronta con quello che impieghiamo normalmente? si poteva far meglio? Come vedremo in seguito l'indicatore più importante in informatica è il tempo di calcolo, non misurato in secondi perché ciò dipende dal calcolatore e dal linguaggio impiegati, ma in numero di operazioni elementari che il programma deve eseguire in funzione della *dimensione dei dati* d'ingresso: cioè, nel caso della moltiplicazione, in funzione del numero n di cifre con cui sono rappresentati i numeri su cui si opera. Il ragionamento sarà svolto *in ordine di grandezza*, concetto matematico su cui torneremo tra breve. Possiamo però affermare sin da ora che l'algoritmo di Ahmes e il nostro sono equivalenti in questo senso e sorprendentemente le operazioni richieste coincidono se rappresentiamo i numeri in base 2 anziché 10.

La risposta alla domanda se si conoscono metodi di moltiplicazione più rapidi è affermativa, a costo di complicare l'algoritmo. Naturalmente ogni algoritmo deve richiedere almeno tempo proporzionale a n perché tutte le cifre dei fattori devono essere esaminate per produrre il risultato, ma a questa efficienza nessuno è ancora arrivato.

Un problema di confronti

Claude Shannon, padre della Teoria dell'Informazione, propose molti decenni or sono un problema di cui presentiamo qui una versione ridotta, sufficiente per discutere alcuni concetti legati agli algoritmi basati su confronti di dati. Algoritmi che, come vedremo durante il corso, costituiscono uno dei pilastri dell'informatica.

Problema delle dieci monete. Tra dieci monete di identico aspetto potrebbe nascondersene una falsa e pertanto di peso diverso. Disponendo di una bilancia a due piatti per confrontare il peso di gruppi di una o più monete, vogliamo individuare la moneta falsa, se esiste, con non più di tre pesate.[5]

[5]Il problema di Shannon prevedeva dodici monete. È ach'esso risolubile con tre pesate ma l'algoritmo è molto complicato.

È implicito che l'eventuale differenza di peso tra la moneta falsa e le altre è inferiore al peso di ogni moneta, per cui non ha interesse porre sui due piatti numeri diversi di monete perché il piatto che ne contiene di più calerebbe rispetto all'altro senza aggiungere alcuna informazione. È altrettanto implicito che il problema deve essere risolto con non più di tre pesate per ogni possibile disposizione dei dati d'ingresso, cioè *nel caso pessimo*. L'algoritmo che costruiremo ha struttura di *albero* che parte da un *nodo* detto *radice* in cui si dichiara il numero e la posizione iniziale delle monete sui piatti, e si dirama a ogni passo in al più tre altri rami a seconda che cali il piatto di destra, o i piatti rimangano in equilibrio, o cali quello di sinisttra. Dalla radice si raggiungono così altri tre nodi in cui si deciderà una successiva allocazione di monete in funzione dell'informazione sin lì ottenuta. Ogni sequenza di nodi così generata dovrà contenere al massimo tre nodi terminando in una *foglia* che contiene una possibile soluzione del problema in cui la moneta falsa è individuata o si stabilisce che non c'è.[6]

Chi non ha dimestichezza con gli algoritmi non si renderà forse conto che la prima operazione da fare in un problema come questo è *enumerare* tutte le soluzioni possibili, perché a ciascuna di esse dovrà corrispondere una foglia dell'albero. E poiché la struttura su dirama in al più tre rami per nodo (e non è detto che tutti e tre i rami esistano), il numero di foglie presenti dopo h diramazioni è al massimo 3^h. Dunque se le soluzioni possibili sono s dovremo avere $s \leq 3^h$. Numerando le monete da 1 a 10, nel nostro problema le soluzioni sono $s = 21$ corrispondenti ai casi "1 falsa e più leggera" (1L), "1 falsa e più pesante" (1P), poi 2L, 2P,.....,10L, 10P, oppure "la moneta falsa non c'è" (0). Poiché con $h = 3$ pesate si possono aprire fino a $3^3 = 27 > 21$ vie di calcolo il problema è ben posto, ma ciò non vuol dire che l'algoritmo esista. E questo modo di ragionare permette di stabilire anche qualcosa di più.

Procediamo per tentativi per renderci conto di cosa ci aspetta. Proviamo a confrontare tra loro due monete, diciamo la 1 e la 2, nella prima pesata (figura 1). Indicheremo questo confronto con 1/2 nella radice dell'albero, da cui questo si dirameràin tre rami: può calare il piatto destro (indicheremo questa situazione con $1 < 2$), o il piatto sinistro ($1 > 2$), o i due piatti possono rimanere in equilibrio ($1 = 2$). L'informazione raccolta nei nodi raggiunti dai tre rami, ove saranno posti nuovi confronti, è anch'essa indicata nella figura. Poiché nel ramo centrale dell'albero dobbiamo ancore decidere tra diciassette possibili soluzioni e con due pesate successive si aprono al più nove percorsi di computazione, la scelta fatta non può condurre alla soluzione richiesta.

Proviamo allora a iniziare con il confronto tra due gruppi di cinque monete, diciamo 1,2,3,4,5 contro 6,7,8,9,10. L'albero generato inizia come indicato nella figura 2. Il ramo centrale $1, 2, 3, 4, 5 = 6, 7, 8, 9, 10$ termina immediatamente sulla soluzione 0, ovvero la moneta falsa non è presente, ma questo penalizza gli altri due rami da ciascuno dei quali dovranno diramarsi almeno dieci percorsi di computazione, il che è impossibile con i due confronti ancora disponibili Dunque anche questa scelta è

[6] Incontreremo ancora la struttura matematica ad albero nei capitoli successivi, in particolare nell'ultimo ove tratteremo di filogenesi.

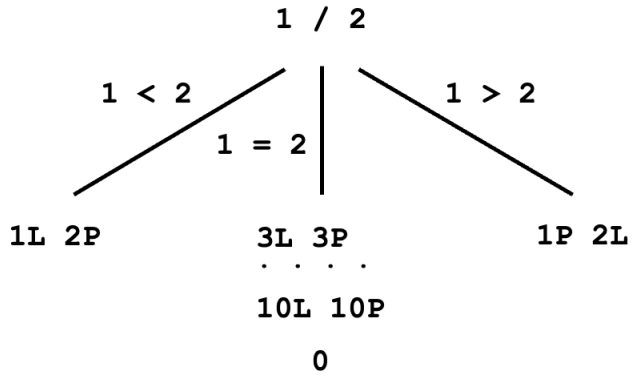


Figura 1. Partenza con due monete

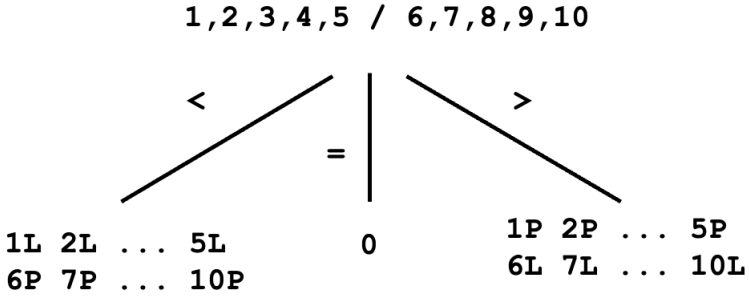


Figura 2. Partenza con dieci monete

inaccettabile. Ciò a cui dobbiamo tendere è dividere a ogni passo l'insieme ivi presente in tre sottoinsiemi *bilanciali*, ovvero con numeri di elementi più vicini possibile tra loro. Come vedremo, il concetto di bilanciamento ha un ruolo fondamentale in molti altri problemi.

Poiché il numero dieci non si divide in tre interi uguali, partiamo confrontando due insiemi di quattro monete come mostrato nell'albero della figura 3 che indica una soluzione. L'albero è indicato per intero, i confronti nei nodi sono in riquadri grigi, l'informazione raccolta su ogni ramo è riportata a lato di questo, le ventuno soluzioni finali sono nelle foglie (in rosso perchi può stampare a colori). Inviterei il lettore a cercare per proprio conto una possibile soluzione partendo dal confronto tra otto monte, prima di esaminare quella proposta. Si dovrà tener conto che:

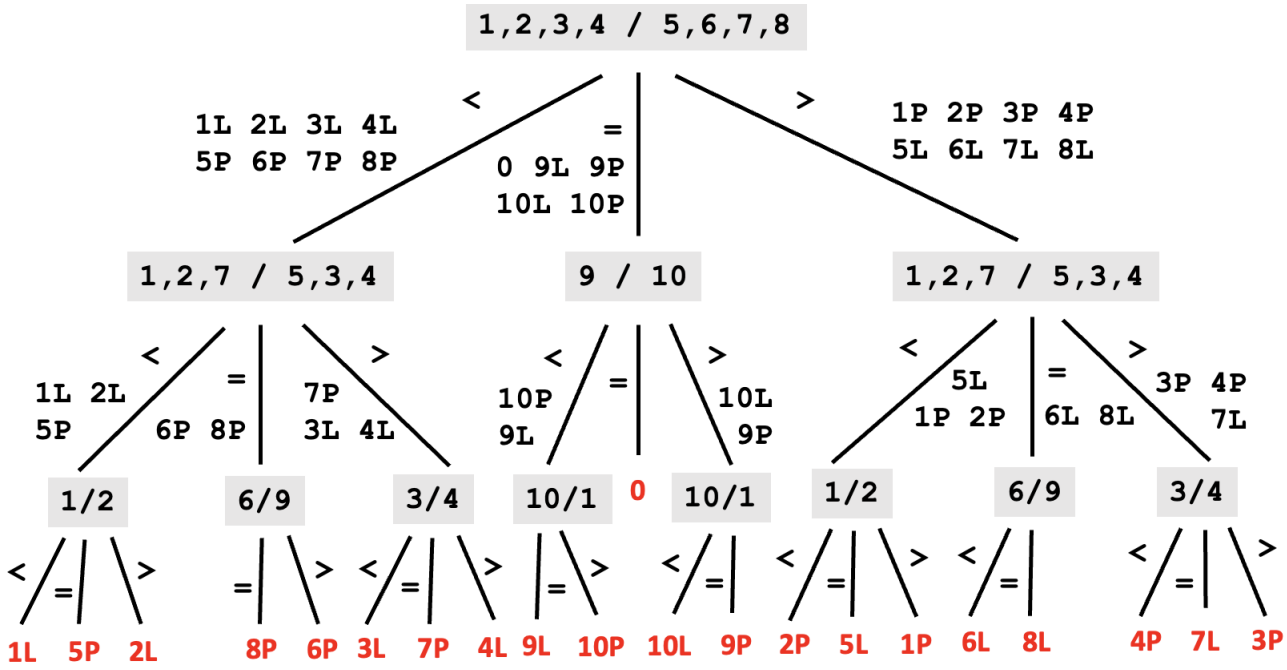


Figura 3. L'algoritmo

- se si verifica uno sbilanciamento tra piatti la moneta falsa esiste ed è una di quelle presenti nei piatti, quindi tutte le altre monete sono certamente buone; questa informazione è implicitamente aggiunta a quelle riportate nell'albero in seguito al confronto, ed è impiegata nella figura 3 nei confronti finali (6/9 e 10/1);
- se in un confronto i piatti si sbilanciano, nel confronto successivo si possono scambiare di piatto alcune monete: se lo sbilanciamento si inverte la moneta falsa è una di quelle scambiate, se lo sbilanciamento rimane uguale quelle scambiate sono certamente buone. Nella figura 3 ciò per esempio è utilizzato nel secondo confronto rispetto al primo, nei rami sinistro e destro.

Come considerazione finale notiamo che sia il pezzo di programma riportato sopra per la moltiplicazione di Ahmes che l'albero di figura 3 sono esempi di coding di un algoritmo: in informatica prevale ovviamente la prima delle due forme che ci permetterà di esprimere algoritmi in modo breve e preciso, eliminando ogni ambiguità.

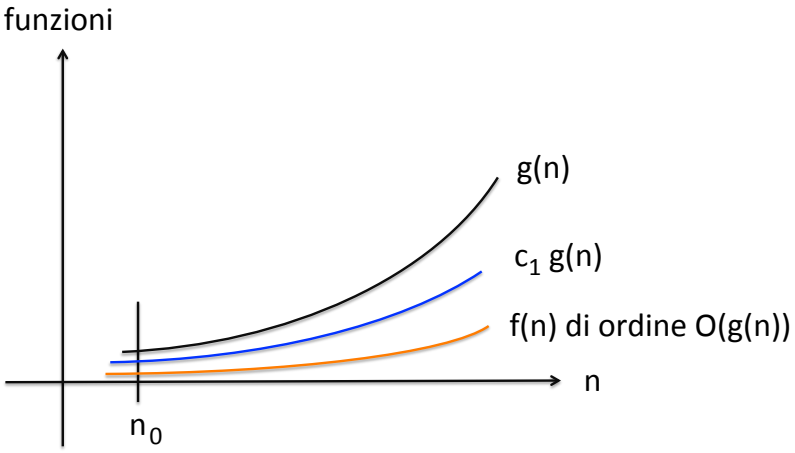
2 ALCUNI CONCETTI MATEMATICI

Ordini di grandezza delle funzioni

In informatica si usa la seguente notazione asintotica per indicare il possibile ordine di grandezza di una funzione. La variabile indipendente n è in genere un intero positivo e indica la "dimensione" dei dati di un problema (per esempio il numero di bit necessario a descrivere i dati). La funzione di cui si studia l'ordine di grandezza è in genere proporzionale al tempo di calcolo (complessità in tempo), oppure alla memoria impiegata oltre a quella necessaria ai dati d'ingresso (complessità in spazio).

Notazione O

$f(n)$ **è di ordine** $O(g(n))$ se esistono due costanti positive c_1, n_0, tali che $0 \leq f(n) \leq c_1\, g(n)$ per ogni $n \geq n_0$.

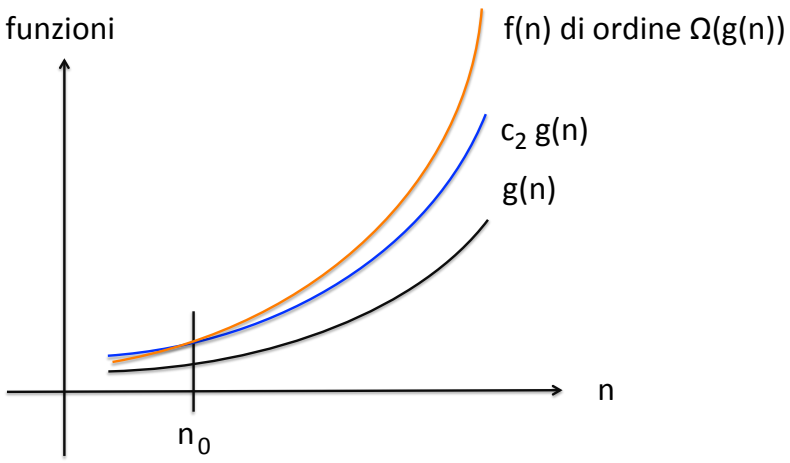


Cioè la funzione $f(n)$ ha un andamento che non sale al di sopra di $g(n)$ al crescere di n, a meno di una costante moltiplicativa. Qui contano solo i termini di ordine massimo. Per esempio $f(n) = 2n^2 - 3n + 1$ è di ordine $O(n^2)$, ma si noti che è anche $O(n^3)$ ecc..

La notazione O si impiega per esempio per indicare il tempo di un algoritmo di cui non si conosce compiutamente il comportamento, ma che si sa che non può superare $g(n)$; oppure che non si comporta allo stesso modo per tutti gli insiemi di dati di dimensione n che gli si presentano, ma per alcuni richiede tempo proporzionale a $g(n)$), per altri meno.

Notazione Ω

$f(n)$ **è di ordine** $\Omega(g(n))$ se esistono due costanti positive c_2, n_0 tali che $0 \leq c_2\, g(n) \leq f(n)$ per ogni $n \geq n0$.

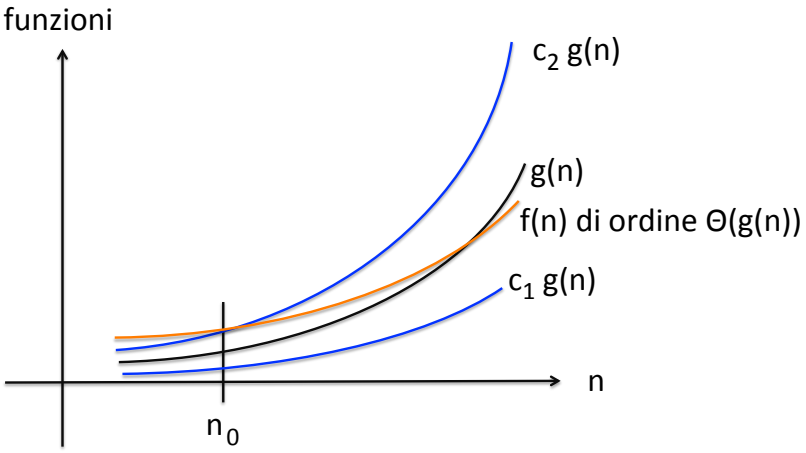


Cioè la funzione $f(n)$ ha un andamento che non scende al di sotto di $g(n)$ al crescere di n, a meno di una costante moltiplicativa. Anche qui contano solo i termini di grado massimo. Per esempio $f(n) = 2n^2 - 3n + 1$ è di ordine $\Omega(n^2)$, ma anche $\Omega(nlogn)$ ecc. La notazione Ω si impiega per esempio per indicare il limite inferiore al tempo di soluzione di un problema, che si applica quindi a tutti i suoi algoritmi di soluzione.

Notazione Θ

$f(n)$**è di ordine** $\Theta(g(n))$ se esistono tre costanti positive c_1, c_2, n_0 tali che $0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ per ogni $n \geq n_0$.

Cioè le funzioni $f(n)$ e $g(n)$ hanno lo stesso andamento al crescere di n a meno di costanti moltiplicative, e contano solo i termini di ordine massimo. Per esempio $f(n) = n^2 - 3n + 1$ è di ordine $\Theta(n^2)$, ma anche $f(n) = 3n^2 - 5$ è di ordine $\Theta(n^2)$ perchés non si considerano le costanti moltiplicative.



La notazione Θ si impiega per esempio per indicare il tempo di un algoritmo di cui si conosce compiutamente il comportamento e che, a pari valore di n, si comporta allo stesso modo per tutti gli insiemi di dati di dimensione n che gli si presentano

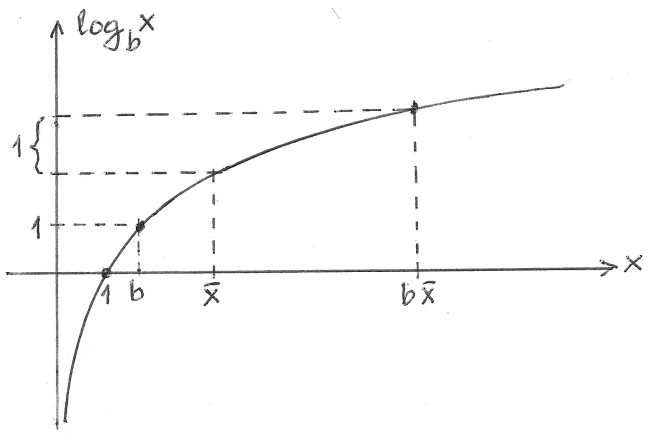
Alcune formule aritmetiche

- $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ per esempio $1+2+3+4+5 = \frac{5\cdot 6}{2} = 15$
- $\sum_{i=0}^{n-1} 2^i = 2^n - 1$ per esempio $1+2+4+8+16 = 32-1 = 31$
- $n! \approx \sqrt{2\pi n}(n/e)^n$, ove $e \approx 2,718$ è la base dei logaritmi naturali:
  questa è la *formula di Stirling* che approssima la funzione fattoriale.
  $n!$ è pari al numero di *permutazioni* di n elementi.
- $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
  questo è il *coefficiente binomiale* che dà il numero di *combinazioni* di n elementi in gruppi di k. Per esempio per $n = 5, k = 2$ le combinazioni dei cinque elementi a, b, c, d, e in gruppi di due sono $5!/(2!3!) = 120/12 = 10$: *ab ac ad ae bc bd be cd ce de*.
- k^n dà il numero di *disposizioni* (con ripetizione) di k elementi in gruppi di n. Per esempio per $k = 2, n = 3$ le disposizioni dei due elementi 0,1 in gruppi di tre sono $2^3 = 8$: 000 001 010 011 100 101 110 111.

Proprietà della funzione logaritmo

1. $y = \log_b x$. Per definizione y è l'esponente che bisogna dare alla base b per ottenere x, ovvero $b^y = x$. Cioè le funzioni logaritmo e esponenziale sono una inversa dell'altra.
2. Considereremo basi intere positive, in particolare $b = 2$ e $b = 10$. È utile aver presente che $2^{10} \approx 10^3$, $2^{20} \approx 10^6$, $2^{30} \approx 10^9$, cioè circa mille, un milione, un miliardo.
3. Dalla definizione 1 risulta con semplici calcoli:
   $\log_b(xy) = \log_b x + \log_b y$; $\log_b(x^y) = y \cdot \log_b x$; $\log_b 1/x = -\log_b x$;
   $\log_b 1 = 0$; $log_b b = 1$; $\log_b(bx) = \log_b x + 1$;
   $\lim_{x\to\infty} \log_b x = \infty$; $\lim_{x\to 0} \log_b x = -\infty$

L'andamento della funzione logaritmo è il seguente:



Per x che tende all'infinito $\log_b x$ tende all'infinito, ma più lentamente di x^e per qualunque esponente $e > 0$ per quanto piccolo (in particolare $0 < e < 1$).

Per cambiare la base di un logaritmo da b ad a si deve moltiplicare per il logaritmo $\log_a b$ tra le basi, cioè:

$$\log_a x = \log_a b \cdot \log_b x.$$

Questa formula è fondamentale. Poiché il termine $\log_a b$ è una costante (cioè è indipendente da x), i logaritmi di x in basi diverse sono proporzionali tra loro. Ciò implica che le basi dei logaritmi non si specificano negli ordini di grandezza perché i diversi logaritmi dello stesso termine differiscono solo per una costante moltiplicativa. Scriveremo quindi $\mathrm{O}(\log x)$ e simili.

Rappresentazione, problemi e algoritmi

La rappresentazione di un oggetto in senso lato (per esempio un'immagine o un insieme di suoni) prende forma in matematica come sequenza di segni da esaminare uno dopo l'altro. Questi segni sono i *caratteri* o *simboli* di un *alfabeto* finito: possono essere segni grafici, lettere, note musicali ecc., e devono essere impiegati in modo che a oggetti distinti corrispondano sequenze distinte.

Per esempio sullo schermo di un computer un'immagine è una matrice di punti colorati organizzati per righe parallele, così piccoli che l'occhio umano non può singolarmente distinguerli ma ovviamente li distingue, li genera e li elabora un insieme di circuiti. Dunque anche l'immagine nel suo complesso è rappresentabile come sequenza di caratteri, ciascuno determinato dalla posizione nella sequenza e dal colore. Il colore a sua volta è rappresentato da una sequenza di caratteri: nel caso dello schermo la sequenza è di ventiquattro bit (caratteri binari) pari a tre byte (gruppi di otto bit): per esempio 11111111 11111111 00000000 è un bel giallo vivo.

Ma quanti colori distinti si rappresentano con ventiquattro bit? Come vedremo $2^{24} > 4.000.000$; un numero enormemente maggiore di quanto sarebbe necessario anche per immagini bellissime, e che trae motivo da altre considerazioni (il byte è un'unità standard nei circuiti commerciali, e ciascuno dei tre byte rappresenta la componente di una tricromia con cui il colore viene formato). È però in genere necessario mantenere basso, o se possibile minimizzare, il numero di simboli richiesti per la rappresentazione di oggetti, e ciò è legato a un principio matematico.

Poniamo che l'alfabeto $\mathcal{A}$ contenga n caratteri, ovvero $|\mathcal{A}| = n$, e indichiamo con N il numero di oggetti da rappresentare. La lunghezza $s(n,N)$ della più lunga sequenza che rappresenta un oggetto dell'insieme (ovvero la "parola" più lunga per dare un nome distinto a tutti gli oggetti) è funzione dei parametri n, N, e il valore di $s(n,N)$ dipende dal metodo di rappresentazione scelto.

Se $n = 1$, cioè $\mathcal{A}$ contiene un solo carattere, i vari oggetti potranno essere rappresentati solo da ripetizioni di quel carattere e almeno una di esse dovrà contenerne N. Se il carattere è 0, i sette nani saranno indicati per esempio con 0,00, . . . , 0000000, secondo una rappresentazione *unaria* che è estremamente sfavorevole come ora vedremo.

Per $n = 2$, convenzionalmente $\mathcal{A} = \{0, 1\}$, la rappresentazione è *binaria*. Per ogni intero $k \geq 1$ le possibili sequenze binarie di lunghezza k sono 2^k e il numero totale di sequenze binarie lunghe da 0 a k è $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$, o $2^{k+1} - 2$ escludendo la sequenza di lunghezza zero cioè vuota. Per rappresentare N oggetti dovremo dunque avere $2^{k+1} - 2 \geq N$; e applicando il logaritmo in base 2 a entrambi i membri della disuguaglianza avremo $k \geq \log_2(N + 2) - 1$. Nella migliore rappresentazione possibile $s(n,N)$ sarà pari al minimo intero k che soddisfa tale relazione, ovvero $s(n,N) = \lceil \log_2(N + 2) - 1 \rceil$, da cui facilmente si ottiene:

$$\lceil \log_2 N \rceil - 1 \leq s(2, N) \leq \lceil \log_2 N \rceil.$$

Dunque $\lceil \log_2 N \rceil$ caratteri binari sono sufficienti (e "quasi" necessari, a meno di 1) per costruire N sequenze differenti e quindi per dare un nome a N oggetti. La rap-

presentazione unaria che richiede almeno una sequenza lunga N mostra dunque uno svantaggio esponenziale rispetto alla binaria che richiede lunghezza logaritmica in N (si ricordi che le funzioni esponenziale e logaritmica sono una l'inverso dell'altra).

Il ragionamento si estende immediatamente a rappresentazioni *n-arie*, convenzionalmente $\mathcal{A} = \{0, 1, ..., n-1\}$ con $n > 2$, per cui si può semplicemente dimostrare che $\lceil \log_n N \rceil - 1 \leq s(n, N) \leq \lceil \log_n N \rceil$, e quindi $\lceil \log_n N \rceil$ caratteri n-ari sono sufficienti per costruire N sequenze differenti. L'esempio più immediato è costituito dai numeri decimali ($n = 10$) per cui con sequenze di k cifre si costruiscono 10^k numeri (per esempio per $k = 3$ si costruiscono i $10^3 = 1000$ numeri da 000 a 999).

In conclusione utilizzando alfabeti di $n \geq 2$ caratteri si possono rappresentare N oggetti con sequenze di $O(\log N)$ caratteri. Tali rappresentazioni sono dette *efficienti* in alternativa alla non efficiente rappresentazione unaria. Si noti che l'influenza della cardinalità n dell'alfabeto per $n \geq 2$ non ha grande impatto sulla lunghezza della rappresentazione perché, come osservato, i logaritmi sono tutti proporzionali tra loro (e infatti la base non è indicata nell'ordine di grandezza). Passando per esempio da base 10 a base 2 il coefficiente di proporzionalità è $\log_2 10 \approx 3.32$, ovvero la rappresentazione binaria dei numeri è circa tre volte più lunga della rappresentazione decimale.

Notiamo infine che quanto fin qui detto non ha relazione con la natura degli oggetti rappresentati ma riguarda solo la possibilità di identificarli con un nome. Ciò non esclude che possa esservi una relazione semantica tra nome e oggetto: l'esempio più evidente è quello dei numeri, ottimizzati al massimo possibile nella numerazione posizionale (o araba) che impieghiamo normalmente.

Studiamo ora un metodo per elencare le sequenze: non è un problema ovvio come potrebbe sembrare. Le sequenze generate con i caratteri di un alfabeto finito sono infinite ma *numerabili*, cioè possono essere poste in corrispondenza biunivoca con i numeri naturali 0, 1, 2 . . . . A tale scopo dobbiamo stabilire un metodo per ordinare le sequenze in modo che ciascuna di esse possa essere raggiunta in un numero finito di passi a partire da l'inizio dell'elenco: per questo si impiega un *ordinamento canonico*. Consideriamo per esempio le sequenze costruite sull'alfabeto $\mathcal{A} = \{a, b, \ldots, z\}$. L'ordinamento canonico è basato sulle regole seguenti:

1. si stabilisce un ordine tra i caratteri dell'alfabeto (nel nostro esempio si userà l'ordinamento alfabetico standard);

2. si ordinano le sequenze per lunghezza crescente e, a pari lunghezza, si segue l'ordine stabilito all'interno dell'alfabeto come in un dizionario.

Nel nostro esempio avremo l'ordinamento:

$a, b, \ldots, z,$
$aa, ab, \ldots, az,\ ba, bb, \ldots, bz,\ \ldots\ldots, za, zb, \ldots, zz,$
$aaa, aab, \ldots, aaz, \ldots\ldots \qquad \ldots\ldots, zzz,$
$\ldots\ldots\ldots$

L'elenco inizia con le sequenze $a, b, \ldots, z$, costituite da un solo carattere, numerate da 1 fino a 26; seguono le aa, ab, ac, ecc. da 27 in poi. Si aggiunge inoltre all'inizio la

sequenza vuota associata allo 0. In questo modo si ottiene una corrispondenza biunivoca tra i numeri naturali e le sequenze: dato qualsiasi numero, a esso corrisponde esattamente una sequenza, e viceversa.

Si noti che l'ordinamento canonico è semplice ma non ovvio: per esempio non potremmo ordinare le sequenze in ordine alfabetico come in un dizionario indipendentemente dalla loro lunghezza, perché l'elenco conterrebbe tutte le infinite sequenze che iniziano con a prima di quelle che iniziano con b: dunque le sequenze che iniziano con b non potrebbero essere a distanza finita dall'inizio dell'elenco e non avrebbero numeri finiti ad esse associati. In matematica si conclude che le sequenze sono *infinite* e *numerabili*: esse costituiscono quindi "il più piccolo" infinito possibile.

Da quanto detto fin qui possiamo affermare che tutta la conoscenza trasmessa dall'uomo è rappresentabile con le parole di un codice costruito su un insieme finito di simboli. E queste parole sono potenzialmente infinite ma numerabili. Occupiamoci ora di problemi e algoritmi per risolverli.

La teoria degli insiemi infiniti nacque alla fine del 1800 per opera di Georg Cantor che dimostrò che esistono infiniti di diverse *cardinalità*, ovvero che non possono essere posti in corrispondenza biunivoca tra loro. L'esempio più famoso è l'infinito dei numeri reali che sono *di più* degli interi, quindi non sono numerabili. La teoria si sviluppò poi nel corso di pochi decenni con conseguenze fondamentali per l'informatica, o meglio per quella parte di essa che va sotto il nome di *algoritmica*.

Scopo di questa disciplina è infatti quello di affrontare la risoluzione di un problema mediante l'esecuzione di una serie di azioni che tipicamente possono essere descritte a una macchina in senso lato; il che richiede di formulare un *algoritmo* e tradurlo in un *programma* per la macchina che operi su un insieme di *dati*. Algoritmo, programma e dati sono descritti con sequenze di simboli e sono quindi potenzialmente infiniti ma numerabili. Questo implica in particolare che i dati su cui i programmi sono chiamati a operare hanno natura *discreta*: per esempio un calcolatore non può lavorare su numeri reali ciascuno dei quali richiederebbe una sequenza infinita di descrizione, ma lavora su approssimazioni finite (tipicamente su troncamenti a un massimo numero di cifre), quindi equivalentemente su interi.

Possiamo sostanzialmente dire che un algoritmo e il suo programma calcolano una *funzione* intera $y = f(x)$ definita sugli interi, perché hanno la cardinalità degli interi sia il *dominio* (valori della x) che il *codominio* (valori della y). Per esempio un programma per ordinare n nomi in ordine alfabetico, ciascuno rappresentato da un gruppo di caratteri di lunghezza prefissata k (per esempio k byte in binario), ha come dato di ingresso una sequenza di n pacchetti di k caratteri e come risultato una sequenza riordinata della stessa lunghezza. Le due sequenze rappresentano rispettivamente i valori di x e y nella funzione di ordinamento $y = f(x)$. Riferendoci a quanto detto sugli ordini di grandezza le due sequenze hanno lunghezza kn e quindi sono di ordine $\Theta(n)$ poiché k è una costante. Un algoritmo di ordinamento è chiamato a calcolare il valore di y per ogni possibile valore di x e la sua complessità sarà funzione di n. Vediamo ora perché la teoria degli infiniti ha influenzato

profondamente la nascita dell'informatica.[7]

Le sequenze fin qui considerate sono in numero infinito, ma sono di lunghezza finita benché a priori illimitata: cioè per qualsiasi intero n si possono costruire sequenze più lunghe di n. Ogni funzione è invece associata a una sequenza infinita. Infatti $y = f(x)$ stabilisce una corrispondenza tra ogni valore di x e un valore di y. Volendola rappresentare in forma tabellare si dovrebbe scrivere un intero y per ciascun intero x: dunque una sequenza infinita di y. Una tabella di funzioni dovrebbe associare una riga a ogni funzione $f_0, f_1, f_2, \ldots$ e una colonna a ogni intero $x = 0, 1, 2, 3, \ldots$, riportando in ogni casella i, j il valore di $f_i(j)$. La tabella ha infinite colonne per x, e infinite righe perché le corrispondenze tra x e y che si possono immaginare sono ovviamente infinite.

Utilizzando il metodo di ragionamento di Cantor si può dimostrare che le funzioni *non sono numerabili*, ovvero esistono più funzioni che interi. Infatti posto che una qualsiasi numerazione $f_0, f_1, f_2, \ldots$ esista, si consideri la particolare funzione definita come:

$$g(i) = f_i(i) + 1, \quad \text{per ogni } i = 0, 1, 2, \ldots$$

cioè la funzione che si ottiene sommando 1 a tutti i valori contenuti nella diagonale principale della tabella ove sono riportati i valori di $f_i(i)$. La funzione $g(n)$ è chiaramente definita in modo legittimo ma non può coincidere con alcuna delle funzioni della tabella perché differisce da ciascuna nel valore riportato nella diagonale, e questo è vero per qualunque numerazione fosse assegnata alle funzioni. Dobbiamo concludere che le funzioni *sono di più* degli interi, ovvero l'insieme delle funzioni è infinito e non numerabile.[8] Ma poiché gli algoritmi sono sequenze finite e quindi numerabili desumiamo un principio fondamentale:

Poiché esistono infinitamente più funzioni che algoritmi, devono esistere infinite funzioni per cui non esiste algoritmo di calcolo.

Tali funzioni, e i problemi che esse rappresentano, sono dette *non calcolabili*, e sorprendentemente sono in numero infinitamente maggiore di quelle calcolabili perché il loro infinito è più vasto di quello degli interi, quindi degli algoritmi.

Nei primi decenni del 1900 questi problemi furono intensamente dibattuti dai logici-matematici: se infatti l'esistenza di funzioni non calcolabili era innegabile, non se ne conosceva nemmeno una. Il problema fu chiuso nel 1936 quando Alonzo Church e Alan Turing posero due definizioni di algoritmo equivalenti tra loro e fino a oggi accettate assieme al primo problema non calcolabile in tali paradigmi, che nel caso di Turing era:

Non può esistere un algoritmo A che decida in tempo finito se un altro algoritmo B termina in tempo finito la sua elaborazione su dati D, per B e D arbitrari.

[7] Chi avesse una vera antipatia per la logica matematica può saltare i due capoversi seguenti e ricominciare la lettura dal principio riportato in grassetto alla pagina seguente (ma è obbligato a crederci).

[8] Lo stesso *argomento diagonale* fu impiegato da Cantor per dimostrare che l'insieme dei numeri reali è più ampio di quello degli interi.. Si può invece dimostrare che i numeri razionali possono essere posti in corrispondenza biunivoca con gli interi, dunque sono numerabili.

Più in generale la lezione di Church e Turing va interpretata come l'impossibilità "inter pares" di decidere del comportamento di un altro se non simulando tale comportamento (ma se B non termina il suo calcolo su D in tempo finito, A non potrà conoscere se B termina o meno attraverso la simulazione di B).

D'ora in aventi ci occuperemo di problemi calcolabili e dei loro algoritmi di risoluzione lasciando all'informatica teorica il compito di indagare sulla non calcolabilià dei problemi. Il che, in un'ottica di integrazione tra l'informatica e la biologia, lascia aperto il formidabile problema di stabilire se il comportamento di un sistema biologico sia o meno dimostratamente imprevedibile.

3 IL PROBLEMA DELLA RICERCA IN UN INSIEME

Ricerca del massimo

Consideriamo un insieme $R = \{r_0, r_1, \ldots, r_{n-1}\}$ di n elementi diversi tra loro per cui è definita una relazione di ordinamento totale indicata con i simboli $<, >$ (scriviamo $a < b$, o $b > a$). Un problema è quello di determinare il *massimo* elemento m di R, ovvero quello $>$ di tutti gli altri.

Ovviamente si tratta di un problema semplicissimo se si dispone di una operazione primitiva di confronto tra elementi, ma vogliamo investigarlo con rigore. In particolare ponendo che il confronto tra due elementi richieda tempo costante, e tale confronto sia l'operazione preminente nei possibili algoritmi che risolvono il problema, valuteremo la complessità di questi come numero di confronti $C(n)$ eseguiti in funzione di n, perché il tempo di calcolo sarà proporzionale a tale numero.

Anzitutto valutiamo un **limite inferiore** a $C(n)$ osservando che, a causa dell'estrema semplicità del problema, potremo presumibilmente calcolare tale limite in modo preciso anziché in ordine di grandezza.

1. Utilizziamo il criterio dell'*albero di decisione* già visto per il problema delle 12 monete. Il problema presente ha $s = n$ soluzioni: $m = r_0, m = r_1, \ldots, m = r_{n-1}$. Ogni confronto genera due possibilità, quindi i confronti successivi generano un *albero binario* che contiene le $s = n$ soluzioni del problema nelle foglie. L'albero binario con n foglie che minimizza il più lungo percorso radice-foglia (relativo al caso pessimo per una soluzione) è quello perfettamente bilanciato in cui tutti i percorsi contengono $\log_2 n$ nodi di diramazione. Ne segue un limite inferiore $C(n) \geq \log_2 n$.

2. Un ragionamento diverso è basato sulla necessità che tutti gli elementi di R entrino in almeno un confronto, altrimenti non potremmo asserire che un elemento mai confrontato non sia massimo. Poiché in Ogni confronto riguarda due elementi almeno $n/2$ confronti sono necessari, ovvero $C(n) \geq n/2$.

3. Un terzo ragionamento è basato sull'osservazione che la determinazione del massimo implica la certezza che gli altri $n - 1$ elementi di R non siano il massimo. Poiché per certificare che un elemento non è il massimo occorre che risulti minore di un altro in almeno un confronto, e da ogni confronto esce esattamente un perdente, sono necessari $n - 1$ confronti per dichiarare $n - 1$ perdenti, ovvero $C(n) \geq n - 1$.

Questi tre limiti inferiori sono tutti legittimi, ma il terzo è superiore agli altri due ed è quindi il più significativo. Concludiamo dunque per ora $C(n) \geq n - 1$, salvo determinare un limite superiore più alto di $n-1$ che lascerebbe aperto il problema di innalzare eventualmente il limite inferiore trovato. Vedremo ora che ciò non accade.

Valutiamo dunque il **limite superiore** a $C(n)$ che, come sappiamo, si ottiene come numero di confronti eseguito dal miglior algoritmo che saremo in grado di trovare. Proponiamo anzitutto un algoritmo in forma *iterativa*, che specifica la sequenza di operazioni in modo esplicito. Ovviamente formuleremo l'algoritmo come programma per un computer.

Poniamo che l'insieme sia memorizzato in un vettore $A[0...n-1]$ con $n > 1$ e supponiamo che il confronto tra elementi si esegua in tempo costante. Un algoritmo iterativo elementare di ricerca del massimo, detto MASSIMO-ITER, consiste nello scandire il vettore A per valori crescenti dell'indice, memorizzando in m, al passo i-esimo, il valore massimo trovato nella porzione del vettore tra $A[0]$ e $A[i]$. Il semplicissimo programma è:

MASSIMO-ITER(A, n, m)
// Ricerca del massimo elemento in un vettore $A[0...n-1]$.
input A, n; **output** m;
$m = A[0]$;
for $(i = 1; i \leq n-1; i++)$ **if** $(m < A[i])$ $m = A[i]$;

Un'analisi elementare dell'algoritmo mostra che questo è corretto ed esegue esattamente $n-1$ confronti. Ovvero un limite superiore è $C(n) \leq n-1$. Poiché questo valore è uguale al limite inferiore trovato prima l'algoritmo MASSIMO-ITER è ottimo e il problema è computazionalmente chiuso. Vogliamo però studiare anche un importante metodo diverso e ugualmente ottimo, basato sul paradigma *ricorsivo* (in alternativa al paradigma iterativo considerato sopra).

In un algoritmo ricorsivo una procedura su dati di dimensione n chiama se stessa su sottoinsiemi di tali dati, finché questi sottoinsiemi raggiungono una dimensione costante (e piccola) per cui il problema è risolto in modo diretto. Tale formulazione, che si presenta a volte come la più spontanea per molti problemi, è anche assai utile perché consente di dimostrare per induzione la correttezza del programma, e si presta a uno studio matematico della complessità in tempo. Per contro diviene però spesso difficile comprendere la sequenza di operazioni effettivamente eseguita dal programma.

Per il problema della ricerca del massimo un algoritmo ricorsivo effettua il calcolo secondo la nota struttura di un torneo a eliminazione diretta: infatti il confronto tra due elementi per determinare il maggiore può essere visto come un confronto sportivo in cui il più forte vince ed elimina l'avversario. Ammettendo per semplicità che sia $n = 2^t$, gli elementi si confrontano in coppie e restano in gara $n/2$ vincitori dopo $n/2 = 2^{t-1}$ confronti; i vincitori si confrontano a coppie, riducendo i partecipanti a 2^{t-2} dopo altrettanti confronti; il torneo prosegue fino alla finale tra $2^1 = 2$ partecipanti, da cui emerge il massimo. Secondo una formula già vista gli incontri sono $\sum_{i=0}^{t-1} 2^i = 2^t - 1 = n - 1$. Dunque anche questo algoritmo è ottimo, anche se è bene notare che gli $n-1$ confronti sono eseguiti in ordine completamente diverso da quelli dell'algoritmo precedente.

Il metodo del torneo è puramente ricorsivo perché il massimo è ottenuto dal confronto finale tra i massimi precedentemente ottenuti tra le due metà dell'insieme R (o tra le due metà del tabellone nella interpretazione sportiva): gli n dati sono cioè divisi in due sottoinsiemi di $n/2$ elementi cui si applica lo stesso algoritmo che "ricorsivamente" divide le due metà dell'insieme in quarti e così via, finché i sottoinsiemi contengono esattamente due elementi e il massimo si determina in altro modo,

cioè con un confronto diretto. Dunque a questo punto iniziano i confronti mentre il resto dell'algoritmo serve a organizzare il calcolo. Il programma MASSIMO-RICOR realizza l'algoritmo specificando come parametri d'ingresso le due posizioni i, j del vettore A che delimitano il sottoinsieme in ogni chiamata della procedura (si ha sempre $i < j$). Il programma è innescato dall'esterno con la chiamata MASSIMO-RICOR$(A, 0, n-1, m)$ sull'intero vettore. La condizione limite in cui un sottoinsieme è ridotto a due elementi si verifica quando $i = j - 1$.

Strettamente parlando l'algoritmo si applica ai valori $n = 2^t$ e richiede una piccola modifica se n non è una potenza di 2. Il caso $n = 2^t$ meglio indica il senso del paradigma ricorsivo.

MASSIMO-RICOR(A, i, j, m)
// Ricerca ricorsiva del massimo in un sotto-vettore di A tra le posizioni i, j su cui lavora la particolare "chiamata" della procedura. Il calcolo nell'intero vettore A è innescato dai valori iniziali $i = 0$, $j = n - 1$, comunicati dall'esterno.

input A, i, j; **output** m;
if $(j - i == 1)$ {**if** $(A[i] < A[j])$ $m = A[j]$ **else** $m = A[i]$;
return m};
$k = \lfloor (i + j)/2 \rfloor$; *//Calcolo dell'indice di mezzo tra i e j*
MASSIMO-RICOR(A, i, k, m_1);
MASSIMO-RICOR$(A, k + 1, j, m_2)$;
if $(m_1 < m_2)$ $m = m_2$ **else** $m = m_1$;

La correttezza dell'algoritmo si dimostra impiegando il principio d'induzione. La base è per $n = 2$ (quindi $i = 0, j = 1$): in questo caso il massimo m è individuato dalla frase **if** iniziale. Per $n = 2^t$ con $t > 1$, poniamo che la procedura calcoli correttamente il massimo per ogni valore $n' < n$: si ha quindi per ipotesi che le due chiamate ricorsive di MASSIMO-RICOR calcolano correttamente i massimi m_1, m_2 nelle due metà del vettore e il calcolo di m è quindi corretto come mostra la frase **if** finale.

Il numero $C(n)$ di confronti tra elementi è soggetto alla seguente *relazione di ricorrenza* che si ottiene immediatamente da un esame del programma:

$$C(n) = 1, \qquad \text{per } n = 2;$$
$$C(n) = 2\,C(n/2) + 1, \quad \text{per } n > 2.$$

Infatti se $n = 2$ il programma esegue un solo confronto senza che le chiamate ricorsive entrino in azione. Per $n > 2$ la procedura richiama se stessa su due insiemi grandi $n/2$ (onde il termine $2\,C(n/2)$) e poi esegue un confronto nella frase **if** finale. Per esprimere esplicitamente $C(n)$ come funzione di n possiamo effettuare lo sviluppo seguente, ove la formula di $C(n)$ è via via applicata a insiemi di dimensioni 1/2, 1/4, 1/8 ecc. su cui operano le chiamate ricorsive, fino a un insieme finale di due elementi cui si applica il valore di $C(2) = 1$. Ponendo $n = 2^t$ si ha:

$$
\begin{aligned}
C(2^t) &= 2\,C(2^{t-1}) + 1 \\
&= 2\,(2\,C(2^{t-2}) + 1) + 1 \\
&\quad = 4\,C(2^{t-2}) + 2 + 1 \\
&= 4\,(2\,C(2^{t-3}) + 1) + 2 + 1 \\
&\quad = 8\,C(2^{t-3}) + 4 + 2 + 1 \\
&= \cdots\cdots\cdots \\
&\quad = 2^{t-1}(C(2^{t-(t-1)}) + 2^{t-2} + 2^{t-3} + \ldots + 1 \\
&\quad = 2^{t-1} + 2^{t-2} + 2^{t-3} + \ldots + 1 \\
&\quad = 2^t - 1 = n - 1.
\end{aligned}
$$

Otteniamo anche in questo caso il limite superiore $n-1$ per $C(n)$: anche questo algoritmo è ottimo benché completamente diverso dal primo.

Esercizio 1. Dato un insieme R di $n = 2^t$ interi positivi diversi tra loro e non ordinati, con $t \geq 1$, si devono determinare il massimo m e il secondo massimo s (cioè il massimo in $R - \{s\}$). L'algoritmo dovrà essere organizzato in due modi:

1. **Facile**. Scandire l'insieme mantenendo a ogni passo i valori di m, s tra gli elementi fin lì esaminati. Calcolare poi il numero di confronti $C(n)$ effettuati dall'algoritmo nel caso pessimo (dovrà risultare $C(n) = 2n - 3$). Qual'è il caso pessimo?

2. **Difficile**. Dividere l'insieme in due metà e risolvere ricorsivamente il problema determinando i valori di m, s per tali metà e confrontandoli per ottenere m, s per l'intero insieme. Calcolare poi il numero di confronti $C(n)$ effettuati dall'algoritmo nel caso pessimo (dovrà risultare $C(n) = 3n/2 - 2$). Qual'è il caso pessimo?

Nota. Non è richiesto di formalizzare i due algoritmi in un linguaggio di programmazione. È sufficiente e più istruttivo immaginare semplicemente come ciascun algoritmo debba essere strutturato: si vedrà che il calcolo di $C(n)$ può essere eseguito anche considerando la sola struttura dell'algoritmo. Scelto così il migliore dei due si potrà affrontare la sua realizzazione pratica sotto forma di un programma.

Ricerca di un elemento

La ricerca di un dato k in un insieme di n dati è uno dei problemi fondamentali nella costruzione di algoritmi. In particolare desideriamo individuare, se esiste, un dato uguale a k perché tale dato normalmente rappresenta un "nome" cui è associata un'informazione aggiuntiva: l'esempio più ovvio è la ricerca di un nome k in una rubrica, per estrarre da questa i dati associati a quel nome.

Proponiamo anzitutto un algoritmo di soluzione che permette di stabilire un limite superiore alla complessità in tempo del problema, formulando l'algoritmo come programma in forma iterativa. Poniamo che gli elementi dell'insieme siano memorizzati nelle celle $A[0], \ldots, A[n-1]$ di un vettore $A[0...n]$ ove la cella $A[n]$ sia usata per controllo, e supponiamo che il confronto tra elementi, le operazioni aritmetiche e di controllo degli indici si eseguano in tempo costante. Studieremo

così il problema tenendo conto di tutte le operazioni eseguite (cioè non soltanto dei confronti).

Un algoritmo *iterativo* elementare di ricerca, detto RICERCA-SEQUENZIALE, consiste nel confrontare in sequenza i dati contenuti in $A[0], \ldots, A[n\text{-}1]$ con k fino a individuarlo se è presente nell'insieme, o raggiungere l'ultimo elemento con esito negativo e stabilire dunque che k non è presente. Ponendo che la risposta dell'algoritmo sia fornita attraverso un parametro intero r, con $0 \leq r \leq n-1$, che indica la posizione di k nell'insieme se tale valore è presente nella cella $A[r]$, e $r = -1$ se k non è presente nell'insieme, l'algoritmo può essere espresso come:

RICERCA-SEQUENZIALE(k, A, r)
// *Ricerca dell'elemento* k *nelle posizioni* $A[0], \ldots, A[n\text{-}1]$ *di un vettore* $A[0...n]$
input k, A; **output** r;
$A[n] = k$;
$i = 0$;
while $(k \neq A[i])$ $i = i + 1$;
if $(i \leq n-1)$ $r = i$ **else** $r = -1$;

La memorizzazione iniziale di k nella cella $A[n]$ garantisce la terminazione del ciclo while su $k = A[n]$ anche nel caso che k non sia compreso nell'insieme: l'algoritmo è in genere riportato in forma un po' diversa nei libri di algoritmica, e la presente ne costituisce una versione migliorata.

RICERCA-SEQUENZIALE si arresta dopo $i+1$ ripetizioni del ciclo se k appare in $A[i]$ con $i \leq n-1$, altrimenti richiede $n+1$ iterazioni se k non è contenuto nell'insieme. Il caso pessimo è dunque quest'ultimo. Poiché tutte le operazioni in ogni iterazione richiedono tempo costante, il tempo $T(n)$ richiesto dall'algoritmo è proporzionale al numero di iterazioni, ovvero $T(n) = O(n)$.

Un metodo alternativo per eseguire la ricerca utilizza il paradigma *ricorsivo*. Un esempio fondamentale è rappresentato dal seguente algoritmo RICERCA-BINARIA che prevede che sia definita una relazione di ordinamento tra i dati (indicata con il simbolo $\leq$), e lavora su un insieme ordinato in ordine crescente. L'algoritmo costituisce una formulazione in termini precisi del metodo usato manualmente per reperire un dato in una rubrica come per esempio l'elenco del telefono. Si inizia in un punto ragionevole dell'elenco (il computer inizia esattamente nel centro); se il dato d ivi contenuto coincide con l'elemento k cercato ci si arresta con successo, altrimenti si ripete l'operazione nella metà sinistra dell'elenco se $k < d$, o nella metà destra se $k > d$, fino a ridursi a un sotto-elenco vuoto che indica che k non è presente. Realizzando l'algoritmo in un programma l'insieme è contenuto in un vettore $A[0...n-1]$ i cui elementi sono disposti in ordine crescente, e il risultato r ha lo stesso significato visto per la ricerca sequenziale.

RICERCA-BINARIA(k, A, i, j, r)
// *Ricerca dell'elemento k in un vettore $A[0...n\text{-}1]$ ordinato in ordine crescente. I parametri i, j indicano gli estremi del sotto-vettore $A[i...j]$ su cui la particolare "chiamata" della procedura deve lavorare: il calcolo è innescato dai valori iniziali $i = 0$, $j = n - 1$, comunicati dall'esterno assieme a k.*

input k, A, i, j; **output** r;
if $(i > j)$ $\{r = -1$; **stop**;$\}$
else $\{m = \lfloor (i + j)/2 \rfloor$;
if $(k == A[m])$ $\{r = m$; **stop**;$\}$
if $(k < A[m])$ RICERCA-BINARIA$(k, A, i, m - 1, r)$
else RICERCA-BINARIA$(k, A, m + 1, j, r)$;
$\}$

Per dimostrare la correttezza della procedura RICERCA-BINARIA notiamo prima di tutto che se k non è presente nel vettore la ricerca giungerà a confrontare k con un sotto-vettore composto da un solo elemento $A[s]$ attraverso la chiamata ricorsiva RICERCA-BINARIA(k, A, s, s, r) che produce $m = s$, nella quale il confronto **if** $(k < A[m])$ innescherà la chiamata ricorsiva RICERCA-BINARIA$(k, A, s, s - 1, r)$ o RICERCA-BINARIA$(k, A, s + 1, s, r)$ che arrestano la procedura restituendo il valore $r = -1$.

Se invece k è contenuto nel vettore la dimostazione procede per induzione su n.

- La base d'induzione è $n = 1$, cioè A contiene il solo valore $A[0] = k$: in questo caso la chiamata iniziale RICERCA-BINARIA$(k, A, 0, 0, r)$ produce $m = 0$ e si chiude con il confronto positivo if$(k == A[0])$.
- Per $n > 1$, posto che la procedura sia corretta per ogni $n' \leq n - 1$, cioè che le due chiamate ricorsive interne alla procedura individuino la posizione di k se presente nel relativo sotto-vettore, la correttezza complessiva deriva immediatamente dall'osservazione che o il test **if** $(k == m)$ individua k, o lo individuerà una delle successive chiamate interne.

Per quanto riguarda la valutazione della complessità in tempo dalla procedura RICERCA-BINARIA, il tempo $T(n)$ da essa richiesto è soggetto alla seguente *relazione di ricorrenza* che si ottiene immediatamente da un esame del programma:

$T(n) = c_1$, con c_1 costante, per $n = 1$;

$T(n) \leq T(n/2) + c_2$, con c_2 costante, per $n > 1$.

Infatti se $n = 1$ il programma richiede un numero costante c_1 di passi per confrontare k con $A[0]$, senza che le chiamate ricorsive entrino in azione. Per $n > 1$ il caso pessimo si ha quando k non è presente in A. In questo caso, oltre a un numero costante c_2 di operazioni di test e calcolo degli indici, si richiama la procedura su un insieme grande $\lfloor n/2 \rfloor$, onde il relativo termine addizionato nella funzione. Per esprimere esplicitamente $T(n)$ come funzione di n possiamo effettuare lo sviluppo

seguente, ove la formula di $T(n)$ è via via applicata a insiemi di dimensioni 1/2, 1/4, 1/8 ecc. su cui operano le chiamate ricorsive, fino a un insieme finale di un unico elemento cui si applica il valore di $T(1)$. Nell'ipotesi semplificativa che sia $n = 2^t$ per un opportuno intero t si ha:

$$\begin{aligned} T(n) &\leq T(n/2) + c_2 \\ &= (T(n/4) + c_2) + c_2 = T(n/4) + 2c_2 \\ &= (T(n/8) + c_2) + 2c_2 = T(n/8) + 3c_2 \\ &= ..... \\ &= (T(n/2^t) + c_2) + (t-1)c_2 = T(1) + tc_2 \\ &= c_1 + tc_2. \end{aligned}$$

Invertendo la relazione $n = 2^t$ abbiamo $t = \log_2 n$, dunque l'algoritmo ha complessità logaritmica $T(n) = O(\log n)$ (si ricordi che la base del logaritmo non appare nell'ordine di grandezza perché i logarimni di un numero calcolati in diverse basi sono correlati tra loro da una costante moltiplicativa). Se n non è una potenza di due il calcolo sopra è approssimato perché i termini $T(n/2^i)$ non sono interi, ma l'ordine di grandezza di $T(n)$ non cambia.

Per renderci conto su un esempio della differenza tra l'efficienza delle ricerche sequenziale e binaria su un insieme di n elementi è sufficiente un calcolo approssimato in modo molto grossolano. Per $n = 10^6$ potremo scrivere che la ricerca sequenziale di un elemento richiede tempo $T_s = h\,n = 10^6\,h$, ove h indica approssimativamente il tempo per calcolare gli indici di due elementi da confrontare ed eseguire il confronto. La ricerca binaria richiederà tempo $T_b = h\,log_2\,n = 20\,h$ perché $10^6 \approx 2^{20}$ e $log_2\,2^{20} = 20$. Quindi su un milione di dati la ricerca sequenziale impiega un tempo circa 50.000 volte maggiore di quella binaria.

Vediamo ora come stabilire un limite inferiore alla complessità in tempo del problema. Se non vi è alcuna relazione tra gli elementi dell'insieme e il modo come essi vengono presentati (in particolare se non sono ordinati) non è possibile per esempio stabilire che k non appartiene all'insieme senza esaminare tutti gli n elementi di questo, il che ci consente di stabilire un limite inferiore di complessità del problema pari a $\Omega(n)$. Poiché la procedura RICERCA-SEQUENZIALE vista all'inizio ha complessità $O(n)$ e quindi eguaglia (in ordine di grandezza) il limite inferiore trovato, possiamo concludere che il semplice ragionamento eseguito per determinare il limite inferiore è del tutto soddisfacente e quella procedura è ottima.

Se invece è possibile eseguire una elaborazione preventiva sull'insieme il ragionamento precedente sul limite inferiore non è più valido. In particolare se gli elementi dell'insieme sono ordinati vale la proprietà transitiva della relazione "$\leq$", secondo la quale se abbiamo per esempio scoperto che $k < A[i]$ sappiamo per transitività che $k < A[j]$ per ogni $j > i$, senza dover eseguire questi confronti. Possiamo quindi determinare un limite inferiore alla complessità del problema considerando il numero minimo $min(n)$ di confronti tra elementi, che devono essere necessariamente eseguiti nel caso pessimo per individuare un dato k in un insieme ordinato, o stabilire che non

vi è contenuto. Il valore di $min(n)$ costituirà così un limite inferiore per il problema, salvo dimostrare che altre operazioni siano più significative in questo caso, ed esista un limite inferiore più alto. A tale scopo utilizziamo un albero di decisione

Il confronto tra k e un elemento $A[i]$ può generare le tre risposte $k<A[i]$, $k=A[i]$, o $k>A[i]$, quindi aprire tre percorsi di computazione nei quali successivi confronti moltilicheranno per tre, a ogni passo, le sitauzioni possibili: in effetti se $k=A[i]$ il percorso si arresta subito con l'individuazione della soluzione per il particolare insieme considerato. Dunque dopo t confronti possono essersi aperte al massimo 3^t vie diverse, al termine delle quali saranno allocate le diverse soluzioni del problema per ogni possibile valore dei dati d'ingresso. Notando che:

- le possibili soluzioni distinte del problema sono $s = n+1$ e cioè: k corrisponde al primo, secondo, ... n-esimo dato dell'insieme, o k non è contenuto nell'insieme;
- come sappiamo ciascuna soluzione deve trovarsi all'estremo di un percorso di compitazione eseguito da un qualsiasi algoritmo di soluzione del problema;
- tali estremi sono al massimo 3^t e deve risultare $3^t \geq s$, quindi $3^t > n$;

applicando il logaritmo a base 3 nella disuguaglianza $3^t > n$, e ricordando che t è un numero di confronti e deve essere approssimato come intero, risulta $t \geq \lceil \log_3 n \rceil$.

Ricordando infine che t è il numero di confronti nel più lungo percorso di computazione (cioè nel caso pessimo: alcuni percorsi potrebbero terminare prima), cioè $min(n) = t$, si ha che $min(n)$ è di ordine $\Omega(\log n)$, che costituisce un limite inferiore alla complessità del problema. Poiché questo limite coincide con il limite superiore $O(\log n)$ stabilito dalla procedura RICERCA-BINARIA, concludiamo che quella procedura è ottima e che il calcolo del limite inferiore basato sui confronti è sufficiente.

Esercizio 2. Controllare su qualche esempio che è valida la seguente proprietà. Si esegua la ricerca binaria in un insieme ordinato R di un elemento k che non vi è contenuto. Detti p e s i due elementi di R che rispetivamente precedono e seguono immediatamente k nell'ordinamento, durante la ricerca k sarà confrontato sia con p che con s. *Dimostrare* la proprietà non è semplicissimo.

4 IL PROBLEMA DELL'ORDINAMENTO

Il problema dell'ordinamento di dati (*sorting* in gergo informatico) consiste nel disporre in ordine "crescente" n elementi tra cui è definita una relazione di ordinamento totale indicata con $\leq$.

Memorizziamo gli elementi da ordinare in un vettore$A[0...n-1]$. Un algoritmo iterativo elementare detto INSERTION-SORT (ordinamento per inserzione), si basa sull'ipotesi che i primi i elementi contenuti nella porzione del vettore tra le posizioni 0 e $i-1$ siano ordinati. Si inserisce l'elemento $A[i]$ tra questi nella posizione che gli compete, ottenendo cosí l'ordinamento fino alla posizione i, e si ripete l'operazione a partire da $i+1$ finché tutti gli n elementi sono stati inseriti. Si inizia con $i=1$ (anziché $i=0$) poiché un sottoinsieme di un solo elemento $A[0]$ si considera ordinato.

INSERTION-SORT(A)
// Ordinamento iterativo di un vettore $A[0...n-1]$.
for $(i=1;\ i \leq n-1;\ i++)$
$\{\ k = A[i]\,;$
// ora si inserisce k nella posizione opportuna della sequenza $A[0...i]$ *che è ordinata fino ad* $A[i-1]$ *confrontandolo k con gli elementi per valori decrescenti dell'indice a partire da* $A[i-1]$.
$j = i-1;$
while $((j \geq 0)$ && $(A[j] > k))$ $\{A[j+1] = A[j];\ j = j-1\};$
// dopo l'istruzione $A[j+1] = A[j]$ *le due celle contengono lo stesso valore; la* $A[j]$ *può essere quindi riutilizzata e si riparte con* $j = j-1$.
$A[j+1] = k\,;\ \}$

Questa procedura è riportata in forme diverse, ma tutte equivalenti, nei libri di algoritmica.

Esercizio 1. Simulare il funzionamento della procedura INSERTION-SORT su un breve vettore scelto a piacere per capire come funziona e studiare i casi ottimo (i dati in ingresso sono già ordinati) e pessimo (i dati sono ordinati in modo decrescente), verificando che nei due casi la procedura richiede rispettivamente tempo $\Theta(n)$ e $\Theta(n^2)$.

Un metodo alternativo per il sorting utilizza il paradigma ricorsivo in cui una procedura su dati di dimensione n chiama sé stessa su sottoinsiemi di tali dati finché questi raggiungono una dimensione costante (cioè indipendente da n) per cui il problema è risolto in modo diretto. Come sappiamo tale formulazione è spesso la più naturale per affrontare un problema, consente di dimostrare per induzione la correttezza del programma e si presta a uno studio matematico della complessità in tempo; ma diviene in genere difficile comprendere la sequenza di operazioni effettivamente eseguita dal programma. Un algoritmo ricorsivo noto come MERGE-SORT (ordina-

mento per fusione) è il seguente: oltre al vettore A, in cui i dati vengono presentati e riordinati man mano, è necessario un vettore di appoggio $B[0...n-1]$.

MERGE-SORT(A, s, d)
// Ordinamento di un vettore A di n elementi tra le posizioni s,d. Nella chiamata iniziale $s = 0, d = n-1$. *Risulterà sempre* $s \leq d$.

if $(s < d)$ {

$m = \lfloor (s+d)/2 \rfloor$; *// m indica il punto medio tra s,d*

MERGE-SORT(A, s, m); MERGE-SORT$(A, m+1, d)$; FUSIONE(A, s, m, d);
}

FUSIONE(A, s, m, d)
// Fusione dei due sottovettori ordinati $A[s...m], A[m+1...d]$ *usando un vettore di appoggio B, per ottenere un sottovettore* $A[s...d]$ *completamente ordinato.*

$i = s;\ j = m+1;\ k = 0;$

while $((i \leq m)\ \&\&\ (j \leq d))$ {

if $(A[i] \leq A[j])$ $\{B[k] = A[i];\ i = i+1\}$

else $\{B[k] = A[j];\ j = j+1\}$;

$k = k+1$ };
// Ora se il secondo sottovettore di A è stato esaurito prima del primo sottovettore, gli elementi rimasti nel primo si trasferiscono nelle ultime posizioni del secondo.

if $(i \leq m)$

$\{j = d-(m-i)$; **for** $(\ ;\ i \leq m;\ i++;\ j++)\ A[j] = A[i]\}$;

// Ora si trasportano in A, a partire dalla posizione s, gli elementi posti in B che sono in totale k.

$i = s;\ j = 0$; **for** $(\ ;\ j \leq k-1;\ i++;\ j++)\ A[i] = B[j]$;

Anche questa procedura è riportata in forme diverse, ma tutte equivalenti, nei libri di algoritmica.

Esercizio 2. Esaminare attentamente la procedura FUSIONE per capire come funziona, eseguendone a mano una simulazione su un piccolo sottovettore (per esempio per $s = 5$ e $d = 8$). Convincersi che richiede tempo $\Theta(d-s)$, quindi tempo $\Theta(n)$ quando lavora sul vettore intero.

Esercizio 3. Eseguire a mano una simulazione di MERGE-SORT su un vettore di otto elementi per comprendere i passi eseguiti da un algoritmo ricorsivo. A tale proposito dare per aquisito che FUSIONE fonde due sottovettori ordinati per formarne uno complessivo, senza ripeterne la simulazione già vista nell'esercizio 2.

Per calcolare il tempo $T(n)$ richiesto da MERGE-SORT notiamo che:

- per $n = 1$ (cioè per $s = d$) la procedura richiede tempo costante b per eseguire il test "if($s < d$)", poi termina: abbiamo dunque $T(1) = b$;
- per $n > 1$ (cioè per $s < d$) la procedura richiede tempo costante c_1 per eseguire il test e calcolare m; poi chiama due volte sé stessa su $n/2$ dati; infine esegue FUSIONE in tempo $\leq c_2 n$ per un opportuna costante c_2 (vedi esercizio 2): abbiamo dunque $T(n) = 2T(n/2) + c_1 + c_2 n \leq 2T(n/2) + cn$ inglobando la costante c_1 nel termine lineare cn, per un opportuno valore costante $c > c_2$.

In conclusione la funzione $T(n)$ soddisfa la seguente *equazione di ricorrenza*:

$$T(1) = b; \quad T(n) \leq 2T(n/2) + cn, \quad \text{per } n > 1;$$

con b, c costanti. Sviluppando con la stessa formula it termine $T(n/2)$ e proseguendo allo stesso modo per i termini via via generati abbiamo $T(n/2) \leq 2T(n/4) + cn/2$, $T(n/4) \leq 2T(n/8) + cn/4$, ecc. Ponendo per semplicità di calcolo che n sia una potenza di 2, cioè $n = 2^t$ e quindi $t = \log_2 n$, otteniamo lo sviluppo:

$$\begin{aligned} T(n) &\leq 2T(n/2) + cn \leq 2(2T(n/4) + cn/2) + cn = 4T(n/4) + 2cn \\ &\leq 4(2T(n/8) + cn/4) + 2cn = 8T(n/8) + 3cn = 2^3 T(n/2^3) + 3cn \\ &\leq .... = 2^t T(n/2^t) + tcn = nT(1) + cn \log_2 n = cn \log_2 n + bn, \end{aligned}$$

ovvero $\boldsymbol{T(n)}$ è di ordine $\mathbf{O(n \log n)}$. Dunque MERGE-SORT è molto più efficiente di INSERTION-SORT che richiedeva tempo quadratico nel caso pessimo, vedi Esercizio 1 (si noti che la funzione $n \log n$ è assai più prossima a n che a n^2).[9]

Vediamo ora di stabilire un limite inferiore alla complessità in tempo del problema determinando il numero minimo di confronti $min(n)$ che devono essere necessariamente eseguiti per ordinare un insieme. Come discusso per il problema della ricerca di un dato in un insieme, $min(n)$ costituisce un limite inferiore: se si potesse dimostrare che altre operazioni, diverse dal confronto, devono essere eseguite in numero maggiore di quello dei confronti si potrebbe stabilire un limite inferiore più alto e quindi più significativo: vedremo però che il limite sul numero di confronti è sufficiente per l'analisi del sorting.

Utilizziamo l'albero di decisione già illustrato per il problema della ricerca. Partendo dalla sequenza iniziale dei dati nel vettore, le $S(n)$ possibili soluzioni del problema sono rappresentate dalle permutazioni di tale sequenza: infatti la sequenza ordinata che si vuole ottenere è una delle possibili permutazioni della sequenza ori-ginale, che sono $n!$. Utilizziamo la *formula di Stirling* che dà un'approssimazione

[9] Nel caso generale in cui n non sia una potenza di due, ovvero $2^{t-1} < n < 2^t$, lo studio di complessità si può eseguire immaginando di allungare il vettore A portandolo a $A[0...2^t - 1]$, allocare i dati da ordinare nelle celle da 0 a $n-1$ e allocare un valore ∞ (cioè il massimo contenibile in una cella di A) in tutte le celle da n a $2^t - 1$. Applicando MERGE-SORT al nuovo vettore si ottiene nelle prime n celle di A l'ordinamento voluto e il calcolo della complessità genera il medesimo risultato in ordine di grandezza.

della funzione fattoriale: $S(n) = n! \approx \sqrt{2\pi n}(n/e)^n$ (vedi dispensa 2.2). Allocando queste soluzioni nelle foglie di un albero di decisione ternario che rappresenta percorsi di computazione di lunghezza massima t abbiamo $3^t \geq \sqrt{2\pi n}(n/e)^n$. Applicando il logaritmo a base 3 ai due membri della diseguaglianza otteniamo:

$$t \geq (1/2)\log_3(2\pi n) + n(\log_3 n - \log_3 e) > n(\log_3 n - \log_3 e)$$

ove il termine $\log_3 n$ è prevalente rispetto al termine costante $\log_3 e$, e dunque t è di ordine $\Omega(n \log n)$.

Ricordando che, nel caso pessimo, t rappresenta il numero di confronti successivi uno all'altro nella computazione più lunga, abbiamo $min(n) = t$, dunque il limite inferiore alla complessità del sorting è $\Omega(n \log n)$. Possiamo perciò concludere che MERGESORT è un **algoritmo ottimo** (mentre INSERTION-SORT non lo è).

5 PROBLEMI POLINOMIALE E ESPONENZIALI

Prendiamo come esempio il problema dell'ordinamento trattato nella dispensa 4, per fare un discorso generale che riguarda la complessità computazionale dei problemi che ci troveremo a dover risolvere.

Abbiamo studiato due algoritmi di ordinamento, MERGE-SORT e INSERTION-SORT, che richiedono tempi $\Theta(n \log n)$ e $\Theta(n^2)$ nel caso pessimo. Indichiamoli brevemente con A_1 e A_2, e mettiamoli a confronto con un terzo algoritmo FOOLISH-SORT qui inventato, il cui nome indica la complessità che ci aspetta. Non essendo particolarmente utile lo descriveremo solo a parole, anche se scriverne un programma non sarebbe così difficile.

In assenza di qualsiasi idea su come eseguire un ordinamento, immaginiamo di generare una a una tutte le permutazioni dei dati d'ingresso, controllandole di volta a volta fino a individuare quella ordinata. Da quanto visto finora FOOLISH-SORT, denominato A_3, richiede tempo $n!$ per generare tutte le permutazioni (nel caso pessimo quella ordinata è l'ultima), e tempo n per verificare l'ordinamento in ciascuna di esse: in complesso tempo $\Theta(n^{3/2}(n/e)^n)$ ricordando l'approssimazione di Stirling del fattoriale (dispensa 4).

A_1 e A_2 sono algoritmi *polinomiali* perché la loro complessità è limitata superiormente da un polinomio nella *dimensione* n dei dati d'ingresso (il tempo di A_1 è limitato superiormente da $n^{1+\epsilon}$ per qualsiasi $\epsilon > 0$ poiché $n^\epsilon > \log n$). A_3 è un algoritmo *esponenziale* perché nell'espressione del tempo n figura (anche) all'esponente. Per renderci conto di quanto ciò possa significare in pratica, poniamo che i tre algoritmi richiedano esattamente tempi espressi in secondi da $T_1 = \frac{1}{100} n \log_2 n$, $T_2 = \frac{1}{100} n^2$, $T_3 = \frac{1}{100} n^{3/2}(n/e)^n$, e indichiamo nella seguente tabella i tempi richiesti dai tre per vari valori di n.

n	4	8	16	32	64	128
T_1	0,08	0,24	0,64	1,60	3,84	8,96
T_2	0,16	0,64	2,56	10,24	40,96	163,84
T_3	0,38	$> 10^3$	$> 10^{12}$	$> 10^{37}$	$> 10^{91}$	$> 10^{215}$

È chiaro dalla tabella che al crescere di n l'algoritmo A_1 si comporta assai meglio di A_2, ma il secondo è comunque accettabile, mentre A_3 richiede tempi inammissibili già per valori di n molto piccoli. La crescita di T_1 e T_2 si può anche studiare facilmente notando che al raddoppiare di n si ha:

$$T_1(2n) = \frac{1}{100} 2n \log_2(2n) = \frac{2}{100} n(\log_2 2 + \log_2 n) = 2T_1(n) + \frac{2}{100} n,$$

$$T_2(2n) = \frac{1}{100}(2n)^2 = 4T_2(n).$$

Per quanto riguarda T_3, la sua impressionante crescita non è gran che influenzata da fattori moltiplicativi, ma dipende dalla presenza del fattore esponenziale

$(n/e)^n$. Qualunque funzione esponenziale presenta infatti una crescita praticamente irragionevole: un ipotetico algoritmo di ordinamento A_4 con tempo $T_4 = \frac{1}{100}2^n$, richiederebbe più di 10^{39} secondi, cioè più di 10^{29} millenni, per ordinare 128 elementi.

Vediamo ora come l'impiego di un calcolatore C' più veloce del calcolatore C su cui sono rilevati i valori della tabella precedente, possa influenzare i valori di n per cui il calcolo risulta ragionevole. Per comprendere l'andamento del fenomeno, immaginiamo che la velocità di C' sia k volte maggiore di quella di C e vediamo come cresce la dimensione dei dati trattabili a pari tempo di elaborazione: calcoliamo cioè il nuovo valore n' ottenuto con gli algoritmi A_1, A_2, A_4 (per semplicità non consideriamo A_3) impiegando C' per un tempo $\bar{t}$, rispetto al valore n ottenuto impiegando C per lo stesso tempo. Come vedremo l'incremento da n a n' dipende dalla complessità dell'algoritmo impiegato: migliore è l'algoritmo, maggiore è il vantaggio ottenuto con un calcolatore più veloce.

Poiché C' è k volte più veloce di C, impiegare C' per $\bar{t}$ secondi equivale in prima approssimazione a impiegare C per $k\bar{t}$ secondi. Per l'algoritmo A_1, impiegando C per n e n' dati, possiamo quindi scrivere:

$$\frac{1}{100}n\log_2 n = \bar{t}, \quad \frac{1}{100}n'\log_2 n' = k\bar{t},$$

da cui si ottiene immediatamente $n'\log_2 n' = kn\log_2 n$. Questa relazione non è facilmente esplicitabile, ma la lentezza di crescita della funzione logaritmo indica che i valori di n' sono quasi k volte i valori di n. Per esempio utilizzando un calcolatore 10 volte più veloce, cioè $k = 10$, per un tempo $\bar{t} = 8,96$ secondi, con A_1 si possono ordinare $n = 128$ dati su C (vedi tabella), mentre per C' avremo $n'\log_2 n' = 10 \cdot 128 \cdot \log_2 128 = 3840$, da cui $n' = 911$ con un fattore di incremento prossimo a 10 (con un algoritmo lineare avremmo ottenuto esattamente $n' = 10\,n$).

Impiegando A_2 possiamo scrivere:

$$\frac{1}{100}n^2 = \bar{t}, \quad \frac{1}{100}n'^{\,2} = k\bar{t},$$

da cui $n' = \sqrt{k}n$. Per $k = 10$ abbiamo $n' = 3,16\,n$ con un incremento di oltre 3 volte, inferiore a quello ottenuto con A_1 che ha complessità minore di A_2, ma sempre consistente. Impiegando un algoritmo polinomiale di complessità n^c, con $c > 2$, avremmo $n' = k^{1/c}n$, con un vantaggio tanto minore quanto più è alto il valore di c, cioè quanto meno efficiente è l'algoritmo.

Da queste osservazioni si deduce che progettare algoritmi di bassa complessità è un vantaggio immediato e futuro. E in questo senso gli algoritmi esponenziali forniscono un comportamento limite. Utilizzando per esempio l'algoritmo A_4 abbiamo:

$$\frac{1}{100}2^n = \bar{t}, \quad \frac{1}{100}2^{n'} = k\bar{t},$$

da cui $2^{n'} = k2^n$; ed estraendo il logaritmo a base 2 da entrambi i membri abbiamo: $n' = n + \log_2 k$. Dunque per algoritmi esponenziali l'incremento di velocità del

calcolatore comporta un beneficio quasi inesistente: non solo l'aumento da n a n' è additivo anziché moltiplicativo, ma il termine aggiunto nella somma è estremamente basso perché ridotto dalla funzione logaritmo. Per esempio motiplicando per $k = 1024$ la velocità del calcolatore si potrebbero ordinare con A_4, a pari tempo, $n' = n + log_2 1024 = n + 10$ dati. **Gli algoritmi esponenziali sono dunque irrimediabilmente inutilizzabili**.

Chiediamoci a questo punto se esistano problemi intrinsecamente esponenziali, tali cioè da richiedere, per essere risolti, tempo sicuramente esponenziale nella dimensione dei dati. Appartengono banalmente a questa famiglia tutti i problemi che richiedono risultati di lunghezza esponenziale: si può per esempio richiedere, dato un insieme di n elementi, che se ne costruiscano le n! permutazioni. Ma problemi così formulati sono sostanzialmente non interessanti perché si chiede di produrre un risultato che non può poi essere realisticamente esaminato.

Il primo problema non banale intrinsecamente esponenziale fu individuato nel 1972 e da allora altri ne sono seguiti, afferenti in genere alla teoria dei linguaggi formali e alla logica matematica. Fortunatamente peró questi problemi non sono in genere importanti in pratiche applicazioni, mentre si incontrano comunemente problemi che richiedono tempo esponenziale nel senso che non siamo capaci di risolverli meglio. Dirigiamo dunque su questi la nostra attenzione.

Per comprendere la natura del fenomeno partiamo dai *problemi decisionali* che chiedono di stabilire se esiste una soluzione con una determinata proprietà, salvo eventualmente produrre tale soluzione in una fase successiva. La risposta è dunque binaria, e ha la forma di affermazione o negazione. Un problema decisionale è detto polinomiale se esiste un algoritmo che stabilisce se esiste una soluzione con la proprietà richiesta in tempo polinomiale nella dimensione dell'input (per la precisione, nel numero di bit necessari a descrivere la particolare istanza del problema da risolvere, o in un parametro proporzionale a quel numero poiché i calcoli saranno eseguiti in ordine di grandezza). Si pone la definizione:

Definizione. $\mathcal{P}$ è l'insieme dei problemi decisionali risolubili in tempo polinomiale.

Consideriamo per esempio un insieme S di n interi positivi e un intero k, e chiediamoci se esistono due elementi di S la cui somma è k. Si tratta di un problema decisionale: lo indicheremo come $P_{\Sigma 2}$ e prenderemo n come dimensione dell'input. Considerando per esempio l'insieme S memorizzato nel seguente vettore A, per $k = 49$ la risposta è affermativa poiché esistono i due elementi $A[1]$, $A[5]$ la cui somma è 49; per $k = 42$ la risposta è negativa:

	0	1	2	3	4	5	6	7
A	22	31	7	47	30	18	15	40

Il problema $P_{\Sigma 2}$ può essere risolto banalmente con il seguente algoritmo iterativo SIGMA2(A, k) che prova la somma di tutte le possibili coppie di elementi. L'analisi di complessità è immediata perché il programma consiste unicamente in due cicli **for** uno nell'altro, per un totale di $O(n^2)$ operazioni.

SIGMA2(A, k)
  for $(i = 0; i \leq n-2; i++)$
    for $(j = i+1; j \leq n-1; j++)$
      if $(A[i] + A[j] == k)$ **return** una coppia esiste;
  return nessuna coppia esiste;

Esercizio 1. Ideare un algoritmo per risolvere $P_{\Sigma 2}$ con $O(n \log n)$ operazioni ordinando inizialmente il vettore A.
Suggerimento. Collocare due puntatori i, j sulle posizioni 0, $n-1$ del vettore ordinato A. Se $A[i] + A[j] = k$ la coppia esiste. Se $A[i] + A[j] > k$ decrementare j di 1. Se $A[i] + A[j] < k$ incrementare i di 1. Iterare i passi precedenti fino a che si incontra una coppia di somma k, o si ha $i = j$ che indica che la coppia non esiste. Convincersi che l'algoritmo è corretto e che la complessità di questa seconda fase (dopo l'ordinamento) è $O(n)$.

$P_{\Sigma 2}$ corrisponde per esempio alla domanda: dati n file di dimensioni note, esistono due di essi che riempiono esattamente un disco di dimensione k? In caso affermativo interessa ovviamente individuare i due file, ma gli algoritmi per deciderne l'esistenza li determinano implicitamente: la difficoltà di questa risposta non è maggiore, in ordine di grandezza, di quella della decisione sulla loro esistenza.

Per quanto riguarda il limite inferiore al tempo di calcolo di $P_{\Sigma 2}$ applichiamo anzitutto il metodo dell'albero di decisione. Il numero di soluzioni del problema è $S(n) = n(n-1)/2+1$, ove $n(n-1)/2$ è il numero di coppie di elementi ciascuna delle quali potrebbe essere una soluzione, e 1 rappresenta il caso in cui nessuna coppia abbia somma k. Considerando i confronti eseguiti dal programma otteniamo un albero di profondità $\geq \log_2 S(n) = \log_2(n^2/2 - n/2 + 1)$, quindi di ordine $\Omega(\log n)$ poiché $\log n^2 = 2 \log n$. Molto più forte è il limite $\Omega(n)$ che deriva dalla semplice osservazione che tutti gli elementi di A devono essere esaminati almeno una volta. Si noti invece che **non tutte le coppie** di elementi devono essere necessariamente esaminate: se per esempio abbiamo esaminato le coppie ($A[i]$,$A[j]$) e ($A[i]$,$A[r]$) stabilendo che $A[i] + A[j] > k$ e $A[i] < A[r]$, non dobbiamo esaminare la coppia ($A[r]$,$A[j]$) la cui somma è sicuramente $> k$.

Dunque siamo ancora in presenza di un gap tra il limite inferiore $\Omega(n)$ e il limite superiore $O(n \log n)$ dell'esercizio 1.

Generalizziamo ora $P_{\Sigma 2}$ nel nuovo problema di decisione P_Σ (noto in gergo come "somma-di-sottoinsieme"): dato un insieme S di n interi positivi e un intero k, esiste un sottoinsieme di S di dimensioni arbitrarie la cui somma degli elementi è k? Nell'esempio numerico precedente, per $k = 92$ la risposta é affermativa poiché $A[2] + A[4] + A[6] + A[7] = 92$.

Per P_Σ, e per molti altri problemi, non si conosce un algoritmo polinomiale e si deve ricorrere a un'enumerazione esponenziale di scelte per risolverli. Nel caso presente un metodo tipico è quello di associare ad A un *vettore di appartenenza* V contenente interi 0, 1 con il significato: $A[i]$ fa parte di una scelta di elementi se

e solo se $V[i] = 1$. Genereremo quindi tutti i possibili vettori V di n elementi e verificheremo se per uno di essi è verificata l'uguaglianza

$$\sum_{i=0}^{n-1} A[i] \cdot V[i] = k \tag{1}$$

nel qual caso la risposta è affermativa (si noti infatti che solo gli elementi per i quali $V[i] = 1$ contribuiscono alla somma). Nell'esempio avremo la soluzione:

	0	1	2	3	4	5	6	7
A	22	31	7	47	30	18	15	40
V	0	0	1	0	1	0	1	1

Una risposta affermativa, come nell'esempio riportato, permette anche di determinare quali siano gli elementi da scegliere nel sottoinsieme attraverso il particolare vettore V che verifica l'uguaglianza (1). Una risposta negativa - il sottoinsieme cercato non esiste - richiede che si generino tutti i 2^n vettori V e se ne provi la non validità, che permette di concludere che il sottoinsieme non esiste. Un algoritmo ricorsivo per generare i vettori V lunghi n contenenti 0, 1 e verificare la relazione (1) per ciascuno di essi è per esempio SIGMA(V, j) riportato sotto. Il vettore A e l'intero k sono dati globali noti al programma. Un intero t, anch'esso globale, noto al programma e da esso modificato nel corso del calcolo, indica quanti vettori di appartenenza sono stati generati fino al momento considerato. Il calcolo viene innescato dalla chiamata SIGMA$(V, 0)$ dopo aver inizializzato t a zero e genera gli elementi di V a partire da $V[0]$. Si noti la posizione delle frasi **return** che interrompono il programma se è stato determinato un sottoinsieme la cui somma è k, o se sono stati esaminati tutti i sottoinsiemi senza trovare soluzione. Naturalmente il programma potrà essere eseguito solo per piccoli valori di n a causa del numero esponenziale di vettori che devono essere generati. Generare tutti i vettori V con un programma iterativo sarebbe notevolmente più difficile.

```
t = 0; // t conta il numero di volte in cui il contenuto di V è stato completato
SIGMA(V, j)
    for (v = 0; v ≤ 1; v++)
        { V[j] = v;
          if (j == n − 1) // il vettore V è stato completato
              { if (Σ_{i=0}^{n−1} A[i] · V[i] == k) return "un sottoinsieme esiste";
                if (t == 2^n) return "nessun sottoinsieme esiste" else t = t + 1;
              }
          else SIGMA(V, j + 1);
        }
```

Esercizio 2 (**Difficile**, ma utile per comprendere il funzionamento di un algoritmo ricorsivo). Simulare a mano i passi dell'algoritmo SIGMA per un esempio a scelta di pochi elementi.

La trattazione matematica dei problemi esponenziali è molto tecnica e piuttosto difficile. Ci limiteremo a riportarne alcuni concetti fondamentali. Per un gran numero di essi, che si presentano come fondamentali in molti campi di applicazione (compresi alcuni aspetti computazionali della biologia) un'informazione addizionale K, detta *certificato*, permette di verificare in tempo polinomiale se i dati D dell'istanza considerata hanno la proprietà desiderata. Tecnicamente ciò significa che esiste un algoritmo di verifica $VER(K, D)$ che controlla in tempo polinomiale l'esistenza di una soluzione per D avente la proprietà richiesta. Nel problema della somma di sottoinsieme il vettore V relativo a una soluzione costituisce un certificato perché, una volta noto, si può verificare la soluzione del problema attraverso la relazione (1) in tempo polinomiale (il calcolo della (1) richiede infatti tempo $\Theta(n)$).

Si pone allora la definizione:

Definizione. $\mathcal{NP}$ è l'insieme dei problemi decisionali verificabili in tempo polinomiale.

P_Σ appartiene dunque a $\mathcal{NP}$.

Mentre la definizione della classe $\mathcal{P}$ è semplice e intuitiva, quella della classe $\mathcal{NP}$ è obiettivamente artificiale e richiede alcune riflessioni perché se ne comprenda la portata. Anzitutto l'*esistenza* di un certificato K non ha alcun effetto sull'algoritmo di soluzione perché K non è noto a priori. Il suo scopo è quello di individuare la classe dei problemi $\mathcal{NP}$: vi sono moltissimi problemi esponenziali che non sono nemmeno certificabili, ma in genere tra essi non vi sono i problemi praticamente più importanti che sono invece in $\mathcal{NP}$.

Il certificato K deve avere lunghezza polinomiale nella dimensione di D altrimenti l'algoritmo VER non potrebbe esaminarlo in tempo polinomiale. Inoltre la definizione di $\mathcal{NP}$ implica che esista un certificato per i dati che hanno la proprietà richiesta: si certifica cioè l'esistenza della proprietà ma non la sua inesistenza. Di nuovo tutto ciò sembra assai artificiale: se un insieme S ammette una certa somma di sottoinsieme il vettore V certifica che tale somma esiste, ma se S non possiede tale somma non è richiesto (e non è comunque sicuro) che ciò sia certificabile in tempo polinomiale. Queste apparenti asimmetrie si spiegano esaminando il modo in cui un problema in $\mathcal{NP}$ può essere risolto, al di là delle definizioni formali.

In sostanza per risolvere un problema in $\mathcal{NP}$ è necessario ricorrere a un metodo enumerativo che esegua un numero esponenziale di prove per verificare se almeno una di queste corrisponde a una soluzione accettabile. Un certificato per il problema è in genere una informazione sulla sequenza di successive scelte che conducono a una soluzione, se essa esiste. Noto il certificato si possono seguire queste scelte raggiungendo direttamente la soluzione in n passi, e verificare quindi in tempo polinomiale che essa esiste. Se però una soluzione non esiste, l'algoritmo deve tentare tutte le

scelte possibili prima di arrestarsi con risposta negativa, e a questo non corrisponde un certificato di lunghezza polinomiale.

Possiamo ora osservare che per ogni problema in $\mathcal{P}$ si può verificare l'esistenza di una soluzione in tempo polinomiale applicando certificato qualsiasi che viene ignorato e risolvendo direttamente il problema. Da ciò segue il risultato fondamentale:

$$\mathcal{P} \subseteq \mathcal{NP}.$$

Non è però mai stato dimostrato se vale il contenimento in senso stretto tra le due classi, cioè se sia effettivamente $\mathcal{P} \neq \mathcal{NP}$. Tale quesito è tra i più importanti problemi irrisolti nella matematica,[10] anche se molti risultati collaterali e oltre quarant'anni di intensa ricerca inducono a credere che $\mathcal{P}$ sia strettamente un sottoinsieme di $\mathcal{NP}$; ovvero esistono problemi in $\mathcal{NP}$, come quello della somma di sottoinsiemi, che con certezza non sono risolubili in tempo polinomiale. In ogni caso finché non sarà dimostrato il contrario dovremo considerare intrattabili i problemi in $\mathcal{NP} - \mathcal{P}$ per cui non conosciamo un algoritmo polinomiale di soluzione.

Come dobbiamo dunque comportarci dinanzi a un problema esponenziale? Scartata l'ipotesi di cercare la soluzione esatta per valori di n che non siano estremamente piccoli, dovremo accontentarci di soluzioni approssimate (per esempio di una somma di elementi di un sottoinsieme di S che sia "circa" uguale a k), ove l'approssimazione richiesta deve essere precisamente definita in relazione al problema considerato. L'importante campo di studi in questo settore ha raggiunto risultati notevoli, ma non possiamo trattarne qui.

Esempi di problemi in $\mathcal{NP}$ per cui non si conosce alcun algoritmo polinomiale di soluzione

1. Biologia molecolare: dato un insieme arbitrario di sequenze di DNA stabilire se esiste una sottosequenza comune di lunghezza assegnata.

2. Manifattura: stabilire se un insieme di pezzi di lamiera possono essere ottenuti tagliandoli da un foglio assegnato disponendoli opportunamente su di esso.

3. Scheduling: dato un insieme di procedure da eseguire su un insieme di macchine, stabilire se tali procedure possono essere distribuite tra le macchine in modo che tutte queste terminino di operare entro un tempo prefissato.

4. Algebra: stabilire se un'equazione algebrica di secondo grado in due variabili x, y, a coefficienti interi, ammette una soluzione in cui x e y hanno valori interi (se l'equazione è di primo grado il problema è in $\mathcal{P}$).

Tutti questi problemi si sanno risolvere solo provando un numero di possibili soluzioni che è esponenziale rispetto ai dati d'ingresso. Si può però facilmente vedere che verificare la correttezza di una soluzione richiede tempo polinomiale.

[10] Vi è un premio internazionale di un milione di dollari per chi riuscirà a risolvere il problema. Si noti però che dimostrare che $\mathcal{P} \neq \mathcal{NP}$ o che $\mathcal{P} = \mathcal{NP}$ potrebbero implicare ragionamenti molto diversi, anche se il motivo di questa asimmetria è troppo complicato per spiegarlo qui.

6 PROGRAMMAZIONE DINAMICA

La programmazione dinamica è un paradigma per la costruzione di algoritmi alternativo alla ricorsività. Esso si usa nei casi in cui esiste una definizione ricorsiva del problema, ma la trasformazione diretta di tale definizione in un algoritmo genera un programma di complessità esponenziale a causa del calcolo ripetuto, sugli stessi sottoinsiemi di dati, da parte delle diverse chiamate ricorsive.

Per comprendere questo punto consideriamo i due algoritmi ricorsivi RICERCA-BINARIA e MERGE-SORT visti nelle dispense 3 e 4. Il primo comprende due chiamate ricorsive nella sua *formulazione*, ma ogni *esecuzione* dell'algoritmo ne richiede una sola a seconda che il dato da ricercare sia minore o maggiore di quello incontrato nel passo della ricerca: in tal modo il calcolo si ripete su un solo sottoinsieme grande $n/2$, poi su un sottoinsieme di questo grande $n/4$, e così via: dividere il sottoinsieme per due fino a ridurlo a un singolo elemento implica, come abbiamo visto, l'esecuzione di $O(\log n)$ passi (si noti l'ordine O e non Θ: infatti l'algoritmo potrebbe terminare prima se il dato cercato si incontra in uno dei primi tentativi).

MERGE-SORT comprende anch'esso due chiamate ricorsive nella sua formulazione e ogni esecuzione dell'algoritmo le richiede entrambe per ordinare le metà "destra" e "sinistra" dell'insieme. Anche in questo caso in $\log n$ cicli i sottoinsiemi si riducono a un singolo elemento, ma poiché entrambi i sottoinsiemi vengono esaminati e il tempo per la loro fusione è lineare, il tempo richiesto, come già visto, è nel complesso $\Theta(n \log n)$.

La grande efficienza di questi due algoritmi rispetto ad altri realizzati in modo meno sofisticato è dovuta alla loro struttura ricorsiva, formulata in modo che chiamate ricorsive diverse agiscano su dati disgiunti: cioè l'algoritmo non è chiamato a ripetere operazioni già eseguite sugli stessi dati. Se ciò non si verifica, l'effetto della ricorsività sulla complessità di un algoritmo può essere disastroso.

Due esempi elementari sono il calcolo dei *numeri di Fibonacci* e il calcolo dei *coefficienti binomiali*. I primi sono definiti come:

$$F_0 = 0,\ F_1 = 1,\ F_i = F_{i-1} + F_{i-2}. \tag{1}$$

Per curiosità questi numeri, già noti nell'antico Egitto, furono introdotti in Europa all'inizio del XIII secolo da Leonardo figlio di Bonaccio (Fibonacci) da Pisa come numero di coppie di conigli presenti in un allevamento ideale in cui i conigli non muoiono mai, ogni coppia ne genera un'altra in un intervallo di tempo prefissato, ma una coppia appena generata deve aspettare il prossimo intervallo prima di essere fertile e generarne un'altra. Detto F_i il numero di coppie presenti al tempo i, l'allevamento è vuoto al tempo iniziale $i = 0$ (dunque $F_0 = 0$); vi si introduce una coppia appena nata al tempo $i = 1$ (dunque $F_1 = 1$) e questa diverrà fertile al tempo $i = 2$ e potrà generarne un'altra a partire dal tempo $i = 3$ secondo una legge per la quale al tempo i il numero di coppie esistenti F_i è pari al numero presente al tempo precedente $i - 1$, più le nuove nate che sono tante quante ne esistevano al tempo $i - 2$ perchè solo queste possono generare al tempo i. Da qui nasce la legge generale

$F_i = F_{i-1} + F_{i-2}$, che genera la progressione:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, ...

Innescata dalla coppia 0,1 la crescita iniziale è piuttosto lenta, ma poi i valori aumentano sempre più velocemente e infatti la loro crescita è esponenziale in i: si può infatti dimostrare che risulta $F_i \approx \frac{1}{\sqrt{5}}\Phi^i$, ove $\Phi = \frac{1+\sqrt{5}}{2}$ è un numero irrazionale pari a circa 1,62 noto in matematica come *sezione aurea*.

Si potrebbe utilizzare l'espressione (1) per formulare direttamente un algoritmo ricorsivo che, per un arbitrario ingresso i, calcola F_i come:

```
FIB(i)
// Calcolo ricorsivo dell'i-esimo numero di Fibonacci generato attraverso
   il costrutto return
   if (i == 0) return 0;
   if (i == 1) return 1;
   return ( FIB(i − 1) + FIB(i − 2) );
```

Il calcolo di F_n si innesca con la chiamata esterna FIB(n). Detto $T(n)$ il tempo richiesto dall'algoritmo avremo:

$T(0) = c_0,\ T(1) = c_1,$ con c_o e c_1 costanti;

$T(n) = T(n-1) + T(n-2) + c_2,$ con c_2 costante, per $n > 1$.

Possiamo semplificare il calcolo notando che $T(n) > 2T(n-2)+c_2$ da cui otteniamo:

$$\begin{aligned} T(n) &> 2\,T(n-2) + c_2 > 2\,(2\,T(n-4) + c_2) + c_2 = 2^2\,T(n-4) + 3\,c_2 \\ &> 2^2\,(2\,T(n-6) + c_2) + 3\,c_2 = 2^3\,T(n-6) + 7\,c_2 > \ldots \\ &> 2^k\,T(n-2k) + (2^k - 1)c_2 > \ldots \end{aligned}$$

da cui per n pari otteniamo $T(n) > 2^{n/2}T(0) + (2^{n/2} - 1)c_2 = 2^{n/2}(c_0 + c_2) - c_2$, e per n dispari otteniamo $T(n) > 2^{(n-1)/2}T(1) + (2^{(n-1)/2} - 1)c_2 = 2^{(n-1)/2}(c_1 + c_2) - c_2$. In ogni caso $T(n)$ è limitato inferiormente da una funzione esponenziale in n, quindi l'algoritmo FIB è esponenziale.

Questo fenomeno è probabilmente inaspettato per chi non abbia dimestichezza con la ricorsività degli algoritmi. Esso dipende dal fatto che le due chiamate ricorsive sono risolte con calcoli indipendenti tra loro, per cui il calcolo di FIB($i-1$) deve essere completato prima di iniziare il calcolo di FIB($i-2$), e il fenomeno si ripete in ogni chiamata successiva. Poiché il calcolo di F_{i-1} richiede per ricorrenza il calcolo di F_{i-2}, questo sarà effettuato per tale scopo, ma il suo valore non sarà più disponibile nella successiva chiamata ricorsiva FIB($i-2$): quindi F_{i-2} sarà calcolato due volte. Esaminando le chiamate successive possiamo renderci conto che F_{i-3} viene calcolato tre volte, F_{i-4} viene calcolato cinque volte, e in genere F_{i-k} viene calcolato un numero F_{k+1} di volte, pari cioè proprio a un numero di Fibonacci. Per esempio l'algoritmo ripete il calcolo di $F_{n-n} = F_0$ un numero F_{n+1} di volte: valore, come

abbiamo visto, esponenziale in n. E qui interviene la programmazione dinamica che si propone di conservare valori già calcolati per utilizzarli di nuovo se ciò è richiesto.

Secondo questo paradigma il calcolo di F_n è efficientemente eseguito con un algoritmo iterativo che costruisce la successione dei numeri di Fibonacci fino al punto n, ove ciascun elemento è direttamente calcolato come la somma dei due precedenti. Poiché ogni elemento della successione si calcola in tempo costante mediante un'addizione, il tempo complessivo è di ordine $\Theta(n)$.[11]

Un problema simile è quello del calcolo dei coefficienti binomiali definiti per due interi $0 \leq m \leq n$ come:

$$\binom{n}{m} = 1, \quad \text{per } m = 0 \text{ e } m = n;$$
$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}, \quad \text{per } 0 < m < n. \tag{2}$$

Per un insieme arbitrario, $\binom{n}{m}$ è il numero di combinazioni di n elementi presi in gruppi di m. Per esempio per l'insieme di cinque elementi $\{a, b, c, d, e\}$ vi sono i $\binom{5}{2} = 10$ gruppi di due elementi : $ab, ac, ad, ae, bc, bd, be, cd, ce, de$. Come noto i coefficienti binomiali appaiono nel *triangolo di Tartaglia*, che si può rappresentare in un'area triangolare all'interno di una matrice M di dimensioni $(n+1) \times (n+1)$, ove i valori di n e m sono relativi rispettivamente alle righe e alle colonne. Per $n = 5$ la M risulta:

	0	1	2	3	4	5
0	1					
1	1	1				
2	1	**2**	1			
3	1	**3**	**3**	1		
4	1	**4**	**6**	4	1	
5	1	5	**10**	10	5	1

ove per esempio $\binom{5}{2} = 10$ è l'elemento in riga 5 e colonna 2.

La matrice è calcolata in base alla formula (2): si noti il valore limite uguale a 1 nella colonna zero e nella diagonale, e gli altri valori calcolati come $M[i, j] = M[i-1, j] + M[i-1, j-1]$. Ciò suggerisce immediatamente di calcolare $\binom{n}{m}$ con un algoritmo iterativo ove gli elementi della matrice sono calcolati progressivamente per righe, determinando gli elementi della riga i per addizioni sulla riga $i-1$ fino a raggiungere $M[n, m]$. E in effetti è sufficiente calcolare gli elementi in una zona di larghezza m e altezza $n-1$ come indicato in grassetto nella matrice qui sopra per il calcolo di $\binom{5}{2}$ (gli 1 nella colonna 0 e nella diagonale sono noti e non vanno calcolati). Poiché ogni elemento della matrice si calcola in tempo costante, l'algoritmo richiede

[11] Per n grande la cosa non è così semplice perché il numero di cifre di F_n cresce con n e non può quindi, da un certo punto in poi, essere contenuto in una sola cella di memoria. Si noti anche che per il calcolo di F_n si può usare la formula approssimata $\frac{1}{\sqrt{5}}\Phi^n$ con opportune correzioni per farla convergere sull'intero richiesto.

tempo proporzionale a $m(n-1)$, cioè di ordine $\Theta(nm) = O(n^2)$ ricordando che $m \leq n$.

Anche questo è un esempio di programmazione dinamica: nato dalla definizione ricorsiva (2), l'algoritmo calcola progressivamente i valori di $M[i,j]$ mantenendoli memorizzati fino al loro impiego per valori successivi degli indici. Uno spontaneo algoritmo ricorsivo avrebbe invece la forma:

BINOMIALE(i,j)
// *Calcolo ricorsivo di* $\binom{i}{j}$*, generato attraverso il costrutto* **return**
if $(j == 0)$ **return** 1;
if $(i == j)$ **return** 1;
return (BINOMIALE$(i-1,j)$ + BINOMIALE$(i-1,j-1)$);

ove il calcolo di $\binom{n}{m}$ si innesca con la chiamata esterna BINOMIALE(n,m). Detto $T(n,m)$ il tempo richiesto da questo algoritmo avremo:

$T(n,0) = c_0,\ T(n,n) = c_1,$ con c_o e c_1 costanti, per qualsiasi n;

$T(n,m) = T(n-1,m) + T(n-1,m-1) + c_2,$ con c_2 costante, per $0 < m < n$.

Come nel caso dei numeri di Fibonacci possiamo semplificare il calcolo notando che $T(n,m) > 2T(n-1,m-1)$ da cui otteniamo:

$$T(n,m) > 2\,T(n-1,m-1) > 2^2\,T(n-2,m-2) > ... > 2^m\,T(n-m,0) = 2^m c_0.$$

Un tempo, dunque, limitato inferiormente da una funzione esponenziale, per gli stessi motivi discussi per i numeri di Fibonacci.

Applicazioni classiche della programmazione dinamica si incontrano nel confronto tra sequenze di caratteri: problemi nati nell'ambito delle basi di dati e degli editori di testo, e oggi importantissimi nelle applicazioni algoritmiche in biologia molecolare (analisi del DNA ecc). Ne vedremo lo schema nella seconda parte di queste dispense.