Elementi di Informatica Teorica

Marcello D'Agostino

Dispensa n. 2.

Indice

Dagli algoritmi ai programmi

Linguaggi di programmazione

Per essere eseguito da un computer, un algoritmo deve essere trasformato in un *programma* scritto in un *linguaggio di programmazione*.

Linguaggi di programmazione

Un *linguaggio di programmazione* è un linguaggio aritificiale le cui espressioni, a differenza di quelle dei linguaggi naturali, sono costruite secondo una sintassi completamente rigida. Questi linguaggi possono differire sensibilimente gli uni dagli altri sia per le regole sintattiche sia per lo "stile di programmazione" che può appartenere a "paradigmi" diversi. I paradigmi di programmazione più importanti sono la *programmazione strutturata* (Fortran, Algol, Pascal, Ada, C), *programmazione funzionale* (Lisp, Haskell, Miranda), *programmazione logica* (Prolog) e *programmazione orientata agli oggetti* (C++), Python, Java). Lo scopo dei linguaggi di programmazione è quello di costruire un ponte fra la descrizione di "alto livello" di un algoritmo — cioè la sua descrizione informale in termini facilmente comprensibili a

un essere umano — e le istruzioni del *linguaggio macchina* che sono direttamente eseguibili da un computer. Queste ultime sono estremamente basilari e fanno riferimento alle operazioni che un computer esegue direttamente in virtù della sua struttura fisica ("hardware"), cioè delle caratteristiche del processore, etc. Così (i) anche un'istruzione molto semplice, espressa in linguaggio macchina, può richiedere centinaia di istruzioni e (ii) il linguaggio macchina può variare da un tipo di computer a un altro. Per queste ragioni, a partire dalla fine degli anni '50 del XX secolo, sono stati introdotti i "linguaggi di alto livello" le cui istruzioni fondamentali sono molto più vicine alle modalità umane di risoluzione dei problemi e possono essere *automaticamente* tradotte in codice direttamente eseguibile, espresso in linguaggio macchina, per mezzo di un apposito programma detto *compilatore*.

Linguaggio macchina

Linguaggi di alto livello

Le istruzioni fondamentali

Per rappresentare le istruzioni fondamentali di un linguaggio di alto livello che sono comuni (al di là di differenze puramente sintattiche) a tutti i linguaggi appartenenti a una certa "famiglia" o "paradigma" si usa uno *pseudolinguaggio* o *pseudocodice*, cioè un linguaggio di programmazione fittizio il cui scopo è quello di rappresentare gli algoritmi in modo più preciso utilizzando istruzioni che possono essere poi facilmente rappresentate, con le opportune variazioni, in qualunque linguaggio appartenente alla stessa famiglia. Così gli pseudolinguaggi sono linguaggi intermedi fra il linguaggio naturale in cui gli algoritmi sono descritti intuitivamente e un linguaggio di programmazione vero e proprio in cui gli algoritmi devono essere *implementati* per poter essere eseguiti da un computer. In questa dispensa useremo uno pseudolinguaggio molto semplice per mettere in luce le caratteristiche fondamentali dei linguaggi riconducibili al primo dei paradigmi di programmazione principali, la programmazione strutturata.

Pseudolinguaggio

Ecco un semplice esempio di programma scritto in questo pseudolinguaggio:

1: **read** n
2: $N \leftarrow 0$
3: **for** $i \leftarrow 1$ **to** n **do**
4: $\quad N \leftarrow N + i$
5: **end for**
6: write N

Questo algoritmo legge in *input* un intero positivo n e restituisce in *output* la somma dei primi n interi positivi. L'istruzione 2 introduce una variabile N e le assegna il valore iniziale 0 (qui usiamo il simbolo

"$\leftarrow$" invece di ":=" per l'operazione di assegnazione di un valore a una variabile). L'istruzione 3 introduce un'*iterazione* o *loop*, chiedendo di eseguire l'istruzione 4 per tutti i valori di $i = 1, 2, \ldots, n$. L'istruzione 4 assegna alla variabile N a sinistra il nuovo valore ottenuto dal valore corrente di N aumentato del valore corrente di i. Per esempio, alla prima esecuzione del loop, il valore corrente di N è 0 e il valore corrente di i è 1, per cui l'istruzione 4 assegna a N il nuovo valore 1. Alla seconda esecuzione, il valore corrente di N è 1 e quello di i è 2, per cui viene assegnato a N il nuovo valore $1 + 2 = 3$, e così via, fino all'ultima esecuzione in cui il valore corrente di N è $1 + 2 + \cdots + n - 1$, il valore corrente di i è n, e N assume il valore finale $1 + 2 + \cdots + n - 1 + n$.

Nel seguito esamineremo le istruzioni fondamentali sui cui è basata la programmazione strutturata.

Istruzione di assegnazione. Un'istruzione di assegnazione ha la forma seguente:

Istruzione di assegnazione

variabile $\leftarrow$ *espressione*

Un'istruzione di assegnazione consiste nell'assegnare a una *variabile* un certo valore denotato da una certa *espressione*. Per esempio se abbiamo una variabile N che può assumere come valori numeri interi, tipiche istruzioni di assegnazione sono le seguenti:

$N \leftarrow 0$
$N \leftarrow N + 1$

Nella prima viene assegnato a N il valore 0, nella seconda il valore corrente della variabile N viene aggiornato assegnando ad essa il nuovo valore dato dal valore corrente aumentato di 1. (Così se le due istruzioni vengono eseguite una dopo l'altra, alla fine il valore di N sarà uguale a 1.)

Istruzione IF-THEN-ELSE. Un'istruzione IF-THEN-ELSE ha la forma seguente:

Istruzione IF-THEN-ELSE

if *condizione* **then**
    blocco1
else
    blocco2
end if

il *blocco1* è una sequenza di istruzioni che vengono eseguite se è verificata la *condizione*; altrimenti, se questa non è verificata, vengono eseguite le istruzioni nel *blocco2*. Ecco un esempio:

if $n \mod 2 = 0$ **then**
    $m \leftarrow n/2$
else
    $m \leftarrow (n-1)/2$
end if

In questo esempio, se il valore corrente di n è un numero pari, viene assegnato a n il nuovo valore $n/2$, altrimenti (cioè se n è dispari) viene assegnato il valore $\frac{n-1}{2}$.

Ciclo WHILE. Un ciclo WHILE ha la forma seguente: Ciclo WHILE

while *condizione* **do**
    blocco
end while

Se la *condizione* risulta soddisfatta, allora viene eseguito il *blocco*. La condizione viene controllata *prima* di ogni iterazione. L'iterazione si ferma quando la *condizione* non è più vera.

Ecco un esempio:

$F \leftarrow 1$
$n \leftarrow 5$
while $n > 0$ **do**
    $F \leftarrow F \cdot n$
    $n \leftarrow n - 1$
end while

Ciclo REPEAT-UNTIL. Un ciclo REPEAT-UNTIL ha la forma seguente: Ciclo REPEAT-UNTIL

repeat
    blocco
until *condizione*

Le istruzioni del *blocco* vengono eseguite se la condizione*non* risulta soddisfatta. La condizione viene controllata *dopo* ogni iterazione. L'iterazione si ferma quando la *condizione* diventa vera. Ecco un esempio:

$X \leftarrow 5$
$Y \leftarrow 0$
repeat
    $Y = Y + X$
    $X \leftarrow X - 1$
until $X = 0$

ciclo FOR. Un ciclo FOR può avere una delle due forme seguenti: Ciclo FOR

for *variabile* $\leftarrow$ *espressione1* **to** *espressione2* **do**
    blocco
end for

Come abbiamo già visto nell'esempio all'inizio di questo paragrafo, nel ciclo FOR le istruzioni del *blocco* vengono eseguite la prima volta assegnando alla *variabile* il valore iniziale denotato da *espressione1*. Poi vengono rieseguite incrementando ogni volta di 1 il valore della variabile fino a che quest'ultima non raggiunge il valore finale denotato

da *espressione2*. A questo punto le istruzioni vengono eseguite per l'ultima volta e si esce dal ciclo. Per esempio:

for $i \leftarrow 1$ **to** n **do**
    $sum \leftarrow sum + i$
end for

Qui nella prima istruzione la variabile *sum* riceve inizialmente il valore 0. Poi ogni esecuzione dell'istruzione nel ciclo FOR aggiorna il valore corrente di questa variabile come indicato, e cioè aggiungendo al valore corrente di *sum* il valore di i, partendo da 1 e incrementandolo di 1 a ogni iterazione fino a che i non raggiunge il valore n:

$$sum \leftarrow 0$$

$$sum \leftarrow 0 + 1 \; [i = 1]$$

$$sum \leftarrow 0 + 1 + 2 \; [i = 2]$$

$$sum \leftarrow 0 + 1 + 2 + 3 + 3 \; [i = 3]$$

$$\vdots$$

$$sum \leftarrow 0 + 1 + 2 + 3 + \cdots + n - 1 + n \; [i = n]$$

Una variante del ciclo FOR è la seguente:

for all *variabile*: *condizione* **do**
    blocco
end for

dove l'istruzione viene eseguita per tutti i valori della *variabile* che soddisfano la *condizione*. Per esempio:

for all n: n è primo $\wedge$ $n \leq k$ **do**
    blocco
end for

Qui le istruzioni nel *blocco* vengono eseguite per ciascun numero primo minore o uguale a un dato valore k.

Istruzioni di lettura/scrittura

Istruzioni di lettura/scrittura

read *variabile*

Esempio: **read** x

In questa istruzione alla variabile viene assegnata l'espressione inserita dall'utente mediante il dispositivo standard di input (per esempio la tastiera).

write *espressione*

Qui l'*espressione* viene inviata al dispositivo standard di output (per esempio, il monitor). Notate che, in un istruzione come "**write** N" se l'*espressione* è una variabile viene scritto il *valore* corrente della variabile, non la variabile. Se vogliamo scrivere una stringa di simboli, dobbiamo metterla fra virgolette, come in:

write "hello"

che scrive la parola "hello". Se invece l'isttruzione fosse:

write hello

senza le virgolette, "hello" sarebbe interpretata come una *variabile* e il computer cercherebbe di inviare al dispositivo di output il valore corrente di questa variabile.

Procedure

Dichiarazione e invocazione di procedura. Come abbiamo visto, nella programmazione strutturata un algoritmo può "invocare" l'esecuzione di un altro algoritmo — detto anche "procedura" o "subroutine" — per eseguire una certa istruzione. In un linguaggio di programmazione questo richiede che venga prima definita con precisione la procedura

- assegnandogli un *nome*,
- speficando la lista dei suoi *parametri*, ovvero le variabili che rappresentano i dati di input,
- specificando il *programma* — inteso come blocco di istruzioni — che deve essere eseguito ad ogni invocazione della procedura.

Questa è quella che si chiama *dichiarazione* della procedura. Il formato generale di una dichiarazione di procedura è il seguente:

procedure NOME PROCEDURA(parametri)
    Programma
end procedure

Per esempio, la procedura per generare tutti i numeri primi compresi fra due interi positivi dati, che abbiamo illustrato nella Dispensa n. 1 di questo modulo e che è nota come "Crivello di Eratostene", può essere dichiarata nel modo seguente:

procedure CRIVELLO($j, k \in \mathbb{N}; 2 \leq j < k$)
    $E \leftarrow [j, j+1, \ldots, k]$
    for $i \leftarrow 2$ **to** $\sqrt{k}$ **do**
        $E \leftarrow$ lista ottenuta da E cancellando multipli di i (tranne i)
    end for
    write E
end procedure

Questa procedura restituisce la lista dei numeri primi compresi fra j e k dati e, una volta dichiarata, può essere invocata da un altro algoritmo tutte le volte che è necessario. L'algoritmo per la scomposizione di un numero in fattori primi, che abbiamo illustrato sempre nella Dispensa n. 1 di questo modulo, la invoca due volte. Se quest'ultimo viene esso stesso invocato come procedura da altri algoritmi (per esempio dall'algoritmo per il massimo comun divisore) dichiariamo anch'esso come una procedura che possiamo poi invocare quando è necessario:

procedure SCOMPOSIZIONE($n \in \mathbb{N}; n \geq 2$)
  $N \leftarrow n$
  $F \leftarrow [\,]$
  $I \leftarrow 0$
  while $N > 1$ **do**
    CRIVELLO(I, N)
    read E
    $I \leftarrow$ primo elemento di E
    while $N \mod I = 0$ **do**
      $N \leftarrow N/I$
      $F \leftarrow F + +[I]$
    end while
  end while
  $P \leftarrow$ prodotto di tutti gli elementi di F
end procedure

Questa procedura invoca due volte la procedura CRIVELLO che abbiamo definito prima con dati diversi e può essere a sua volta invocata da altri algoritmi, come quello per il MCD:

1: **Input**: due interi j, k; $\neg(j = 0 \wedge k = 0)$
2: SCOMPOSIZIONE(j)
3: **read** P
4: $P1 \leftarrow P$
5: SCOMPOSIZIONE(k)
6: **read** P
7: $P2 \leftarrow P$
8: $S \leftarrow$ *prodotto dei fattori comuni a* $P1$ *e* $P2$ *con il minimo esponente*
9: **write** S.

In questo algoritmo l'istruzione 8 chiede di eseguire un compito complesso per il quale possiamo definire un'ulteriore procedura, e così via.

Diagrammi di flusso

Oltre che per mezzo delle istruzioni di uno pseudolinguaggio, gli algoritmi possono essere rappresentati in modo più facile da compren-

dere visivamente, anche se meno preciso, dai cosiddetti "diagrammi di flusso". Un *diagramma di flusso* è un sistema di "scatole" rappresentate da figure geometriche collegate da frecce che viene usato per fornire una rappresentazione grafica degli algoritmi. Le figure sono di tre tipi (vedi Fig. 1:

Diagrammi di flusso

- *ovali* per indicare l'inizio e la fine dell'algoritmo;
- *rettangoli* contenenti le istruzioni che devono essere eseguite;
- *rombi* contenenti condizioni dalla cui verifica o meno dipendono le istruzioni successive.

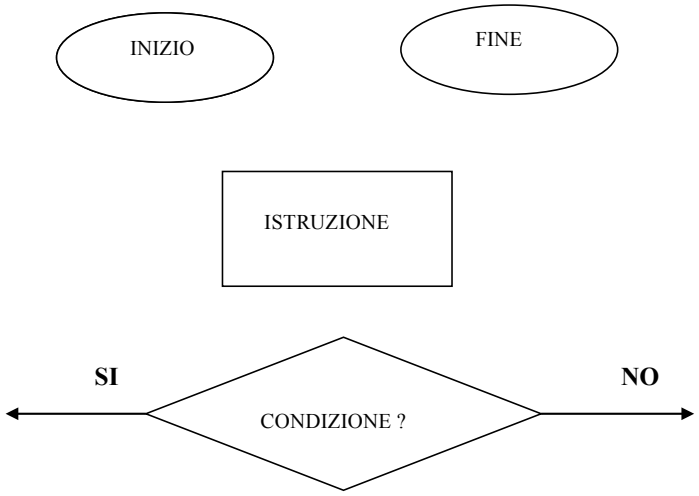


Figura 1: Le "scatole" di un diagramma di flusso

Il diagramma di flusso in Fig 2 rappresenta il semplice algoritmo per riconoscere i numeri primi illustrato nella Dispensa 1 di questo modulo.

Nelle Fig. 3-6 le istruzioni fondamentali discusse nel paragrafo precedente sono rappresentate sotto forma di diagrammi di flusso.

Organizzare l'informazione

La capacità di progettare un buon algoritmo per risolvere un problema dipende in larga misura dalla nostra capacità di *organizzare l'informazione*. I dati che abbiamo a disposizione spesso hanno esplicitamente una *struttura* o possono essere riorganizzati, con un po' di lavoro, in modo da far emergere una struttura nascosta. In informatica, si parla di *tipi di dati* per indicare queste *collezioni strutturate* di

Tipi di dati

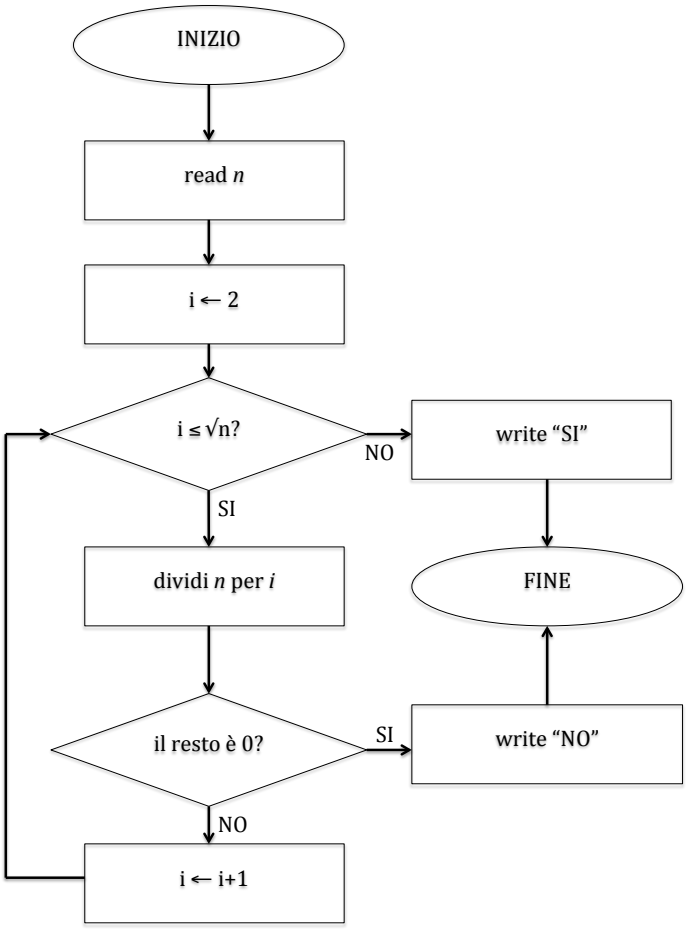


Figura 2: Algoritmo per il riconoscimento dei numeri primi

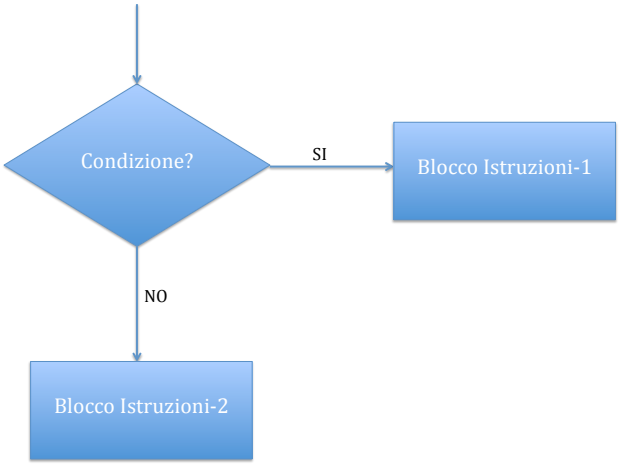


Figura 3: Diagramma di flusso per IF-THEN-ELSE

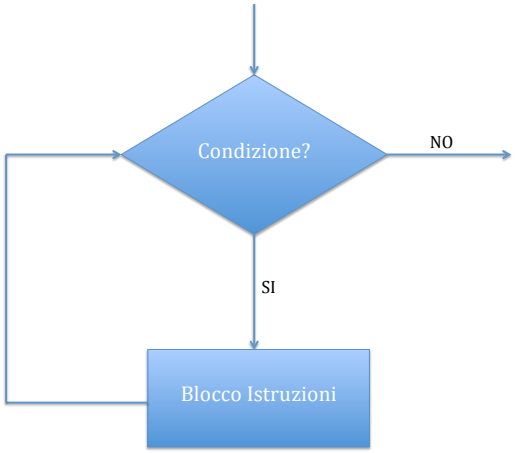


Figura 4: Diagramma di flusso per il ciclo WHILE

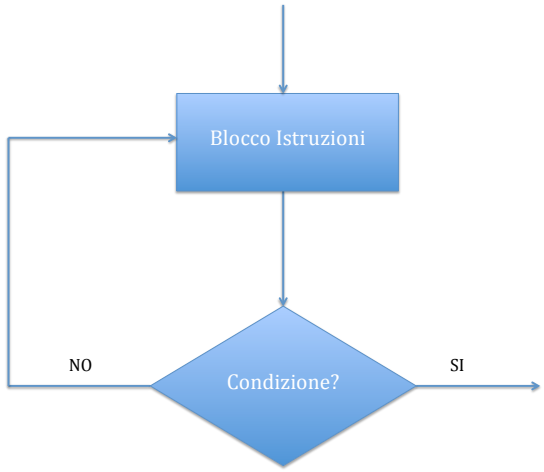


Figura 5: Diagramma di flusso per il ciclo REPEAT-UNTIL

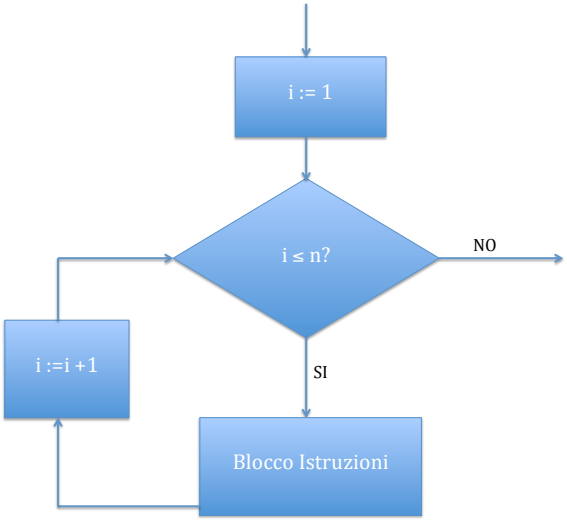


Figura 6: Diagramma di flusso per il ciclo FOR

informazioni che sono i mattoni con cui si costruiscono gli algoritmi e dalla scelta delle strutture più appropriate dipende spesso la loro chiarezza e la velocità con cui essi forniscono risposte alle nostre domande. Sebbene i tipi di dati usati nella risoluzione di problemi siano innumerevoli, la maggior parte possono essere ricondotti a tre tipi astratti fondamentali: le *liste* (o *sequenze*), gli *alberi* e i *grafi*.

Liste

Quando vi iscrivete a un esame online, il vostro nome viene aggiunto a quello delle persone che si sono già iscritte. Quando il professore apre l'elenco degli iscritti quello che vede sono dei nomi scritti *uno dopo l'altro* dall'alto in basso. Questo è un semplice esempio di una *lista* o *sequenza* di nomi. Qui la struttura è data dall'ordine degli elementi e dal fatto che è possibile identificare il primo, il secondo, il terzo, etc. In questo caso, non accade mai che lo stesso nome ricorra più di una volta in una lista, perché il sistema non consente allo stesso studente di iscriversi due volte allo stesso appello. Ma se lo consentisse, potremmo anche avere una lista in cui lo stesso nome ricorre più di una volta in posizioni diverse. Altri esempi di liste sono l'ordine di arrivo di una gara (qui non ci sono ripetizioni) oppure l'elenco dei voti che avete ottenuto agli esami nell'ordine in cui li avete ottenuti (qui invece di solito ci sono ripetizioni), la lista della spesa, etc. Notate che anche le parole di una lingua o le frasi che possono essere costruite con esse sono esempi di liste i cui elementi sono i caratteri dell'alfabeto: la parola "cane" non è altro che la lista costituita dalle lettere "c", "a", "n", "e" nell'ordine da sinistra.

In astratto, possiamo dire che:

Una *lista* o *sequenza* è una collezione di dati che contiene anche informazioni sulla posizione di ciascun dato, in modo che sia possibile dire qual è il primo, il secondo, il terzo, etc.

Liste

Formalmente, una lista di n elementi può essere rappresentata come una *funzione* f che assegna a ciascun numero naturale da 1 a n un elemento di un certo dominio di oggetti D. Nella notazione che abbiamo usato nel paragrafo precedente, una lista viene specificata elencando i suoi elementi fra parentesi quadre separati da una virgola. Per esempio $[1,3,1,4,2]$ rappresenta in termini matematici, la funzione $f : \{1,\ldots,5\} \to \mathbb{N}$ tale che $f(1) = 1$, $f(2) = 3$, $f(3) = 1$, $f(4) = 4$, $f(5) = 2$. Spesso per denotare gli elementi di una lista si usa una singola variabile indicizzata da un numero naturale che rappresenta la posizione di un particolare elemento nella lista. Per esempio data la lista $x = [1,3,1,4,2]$, $x_1 = 1$, $x_2 = 3$, etc. Un ca-

$\mathbb{N}$ è l'insieme dei numeri naturali $1, 2, 3, \ldots$.

so particolare è dato dalla *lista vuota*, che abbiamo denotato con $[\,]$. Un'operazione molto comune con le liste consiste nella loro *concatenazione*. Per concatenare la lista L_1 e la lista L_2 si aggiungono tutti gli elementi di L_2, nell'ordine dato, alla fine di L_1. Abbiamo usato il simbolo "++" per denotare l'operazione di concatenazione di due liste, per cui $L_1 + +L_2$ indica la lista che risulta dalla concatenazione di L_2 a $L1$. Per esempio: $[h, e, l] + +[l, o] = [h, e, l, l, o]$. Ovviamente, la concatenazione della lista vuota con qualsiasi lista L è uguale a L stessa:

$$[\,] + +L = L.$$

Che cos'è una lista?

La concatenazione di liste gode anche della *proprietà associativa*:

$$(L + +L') + +L'' = L + +(L' + +L''),$$

ma *non* gode della *proprietà commutativa*. In generale:

$$L + +L' \neq L' + +L,$$

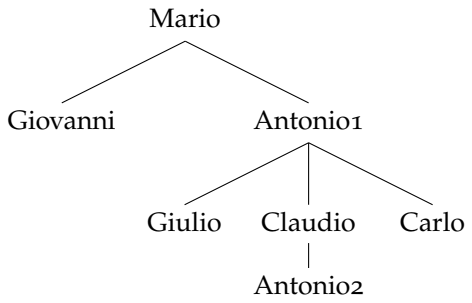
tranne che nel caso speciale in cui le due liste sono identiche.

Alberi

Un altra struttura di grande importanza che abbiamo già incontrato informalmente nelle dispense di Logica sono gli *alberi*. Un esempio familiare a tutti è costituito dagli *alberi genealogici*. Queste sono collezioni di nomi connessi fra loro in modo da mettere in evidenza le discendenze patrilineari. Per esempio:

Alberi

Spesso l'espressione "albero genealogico" è utilizzata in modo improprio per indicare grafici che specificano anche le unioni matrimoniali o extraconiugali, e anche altre informazioni. Ma in questo caso le strutture non sono alberi ma, più in generale, "grafi decorati".



è un albero di nomi che indica i rapporti di discendenza patrilineare. In questo caso, Mario è padre di Giovanni e Antonio; Antonio è padre di Giulio, Claudio e Carlo, e Claudio ha, a sua volta, un solo figlio di nome Antonio. I vari elementi che vengono connessi fra loro nell'albero vengono detti *nodi*. Le caratteristiche delle strutture di questo tipo sono le seguenti (notate che, per convenzione, in matematica gli alberi crescono verso il basso!):

1. ogni nodo ha sempre un *numero finito* (anche nullo) di discendenti immediati (detti *nodi figli*);

2. ogni nodo ha *al massimo* un predecessore immediato, che viene detto il suo *nodo genitore*;

3. c'è un unico nodo che non ha un genitore, che viene detto la *radice* dell'albero.

Un nodo che non ha figli viene detto una *foglia*. Infine, un *ramo* dell'albero è qualunque sequenza di nodi che comincia con la radice e finisce con una foglia. Nel nostro esempio sopra, Mario è la radice, Giovanni, Giulio, Antonio e Carlo solo le foglie, mentre i rami dell'albero sono, rispettivamente, le sequenze di nodi:

[Mario, Giovanni], [Mario, Antonio, Giulio],
[Mario,Antonio, Claudio, Antonio], [Mario,Antonio, Carlo].

Gli alberi sono strutture di grande importanza in tutti i rami della ricerca scientifica e operativa. Sono alberi, per esempio, i sistemi di file che sono memorizzati nei vostri computer. Ogni cartella può contenere diverse cartelle "figlie" e queste, a loro volta possono contenere altre cartelle, e così via. Per ogni cartella è possibile identificare al massimo una cartella che la contiene, cioè una sola cartella "genitore". E c'è una cartella, quella associata al vostro nome utente, che non è contenuta in nessuna cartella, cioè una cartella "radice". Le "foglie" sono i file (per esempio un file di testo o un foglio di spreadsheet) che non sono cartelle e contengono informazioni direttamente utilizzabili. Un "ramo", in questo caso è il percorso completo che dovete seguire per andare dalla cartella radice a un file. E questo percorso completo è il "vero" nome che identifica univocamente un file se volete accedere ad esso dall'esterno della cartella in cui si trova; e.g.:

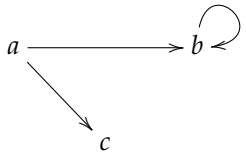
```
DAgostino/Dropbox/myteaching/materiali_didattici/L&C_DEM/
Fondamenti_di_Informatica/handouts/FondInf_settimana2.pdf
```

è il vero nome di questo file nel mio filesystem. Questo nome completo rappresenta il ramo dell'albero che conduce dalla radice (`DAgostino` che è la mia cartella personale) alla foglia (`FondInf_settimana7.pdf`) che è costituita da questo file in pdf.

Grafi

Che cos'è un grafo?

Gli alberi sono casi speciali di strutture di dati più generali: i *grafi*. Abbiamo già usato informalmente i grafi nelle dispense di Logica per descrivere i mondi possibili relativi al linguaggio dell'aggressività. Dovevamo descrivere le relazioni di aggressività fra tre ragazzi, Arabella, Bianca e Carlo, e lo abbiamo fatto disegnando dei diagrammi in cui dai nomi dei ragazzi partivano un numero finito (anche nullo) di frecce che rappresentavano la relazione descritta dalla proposizione aperta "x è aggressivo con y". Per esempio, il diagramma:



rappresenta il mondo possibile in cui Arabella è aggressiva con Bianca e con Carlo, Carlo non è aggressivo con nessuno e Bianca è aggressiva solo con se stessa. Questo diagramma è un tipico esempio di *grafo orientato*. Si dice "orientato" perché la relazione che vuole descrivere ha una direzione (se x è aggressivo con y non è detto che y debba essere aggressivo con x), per cui le frecce non possono essere sostituite da connessioni non orientate.

Ma potremmo anche essere interessati a costruire dei grafi per rappresentare relazioni binarie *simmetriche*, in cui dunque la direzione non ha importanza. Per esempio, se vogliamo tracciare le relazioni di amicizia fra un gruppo di persone — intendendo la relazione di amicizia come una relazione simmetrica (se x è amico di y allora anche y è amico di x come avviene in certi social networks) — possiamo costruire un grafo *non orientato* in cui le connessioni non hanno frecce. Per esempio:

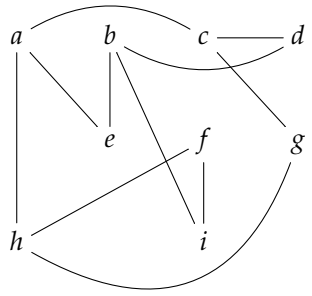


Figura 7: Un grafo non orientato.

Notate che in generale la relazione di amicizia *non è transitiva*. Non necessariamente sono amico di tutti gli amici dei miei amici.

I grafi sono di fondamentale importanza per rappresentare un enorme varietà di dati srutturati: i sistemi di autostrade, le connessioni del trasporto aereo, le relazioni di accessibilità in una rete, le connessioni fra gli utenti di un social network, etc. In generale, quando c'è una *relazione binaria* fra gli elementi di un certo dominio V, cioè una relazione che può esserci o non esserci fra due elementi x e y di V, questa relazione può essere rappresentata da un grafo. Se si tratta di una relazione simmetrica si userà di un grafo non orientato, altrimenti si userà un grafo orientato. Per esempio, se siamo interessati alle relazioni di amicizia su *facebook* useremo un grafo non orientato, perché su facebook la relazione *x è amico di y* è simmetrica. Invece,

su *twitter*, la relazione *x è un follower di y* non è simmetrica, per cui se siamo interessati a questa relazione useremo un grafo orientato.

In termini astratti:

Un *grafo* è una struttura costituita da un insieme V di elementi, detti *vertici*, o anche *nodi* e da un insieme E di coppie non ordinate di elementi di V dette *archi*, o *spigoli*, o anche *connessioni*. In un *grafo orientato*, l'insieme E è costituito da *coppie ordinate*.

Definizione formale di grafo

È chiaro che, se il grafo rappresenta un certa relazione binaria R, l'insieme E delle connessioni non è altro che l'insieme di tutte le coppie di elementi x, y tali che $R(x, y)$ è vera. Come abbiamo visto, se il grafo è non-orientato, una connessione $(a, b) \in E$ viene rappresentata da una linea semplice che unisce i vertici a e b. Se invece il grafo è orientato, una connessione $(a, b) \in E$ è rappresentata da una freccia che parte da a e arriva a b.

Usiamo la notazione (a, b) per denotare ambiguamente la coppia non ordinata costituita dagli elementi a e b se il grafo è non orientato e la coppia ordinata costituita dagli stessi elementi se il grafo è orientato.

Dato un grafo $G = (V, E)$, due vertici a e b si dicono *adiacenti* se $(a, b) \in E$. Si dice *cammino* di un grafo una successione di vertici adiacenti.[1] Se l'ultimo vertice è uguale al primo, il cammino è *chiuso*. Nel nostro esempio in Fig. 7, $[a, e, b, d]$ è un cammino che connette a a d e $[a, c, d, b, e, a]$ è un cammino chiuso. Un grafo si dice *connesso* se per ogni coppia di vertici a e b esiste un cammino che li connette. (Il grafo del nostro esempio in Fig. 7 è un grafo connesso.) Infine si dice *circuito* di un grafo un cammino chiuso in cui tutti i vertici sono distinti (tranne il primo e l'ultimo che sono uguali). Il cammino chiuso $[a, c, d, b, e, a]$ del grafo in Fig. 7 è anche un circuito.

[1] Più precisamente, un cammino è una successione $a_1, \ldots, a_n$ di vertici tali che a_i e a_{i+1} sono adiacenti per ogni i compreso fra 1 e $n - 1$.

Un esempio: il calendario degli esami

Alberi e grafi sono strutture matematiche di importanza centrale in un gran numero di aree della matematica discreta e applicata, come il calcolo combinatorio, la teoria dei linguaggi formali, la teoria delle reti, la teoria degli algoritmi e quella delle basi di dati, ma anche in molti settori delle scienze empiriche, come la biologia molecolare, dove vengono utilizzati intensamente per rappresentare le interazioni nei sistemi biologici. In Economia l'uso dei grafi per rappresentare l'informazione è pervasivo, soprattutto nella teoria dei giochi e nella soluzione di problemi di ottimizzazione. Illustriamo ora l'uso dei grafi per risolvere un problema di grande importanza pratica.

CALENDARIO DEGLI ESAMI: Costruire un calendario degli esami tale che nessuno studente debba sostenere due esami nello stesso giorno.

Per risolvere questo problema, dobbiamo prima capire bene come *rappresentare i dati*. Qui i dati sono: (i) gli esami che possono essere sostenuti in un certo periodo didattico (ii) per ogni coppia di esami distinti, l'informazione se questi esami hanno candidati comuni, cioè se ci sono studenti che hanno diritto a sostenerli entrambi.[2] A questo scopo possiamo costruire un grafo non-orientato come quello in Fig 7. Possiamo usare lo stesso grafo supponendo che a, b, c, etc. siano gli esami che possono essere sostenuti nel periodo didattico in questione e una connessione fra due vertici indichi che ci sono studenti che possono sostenere entrambi gli esami.

[2] Notate che la relazione "ci sono candidati comuni agli esami a e b" (con $a \neq b$) è una relazione simmetrica, ma non è, in generale, una relazione transitiva (visto che alcuni esami sono opzionali).

Il nostro obiettivo è evitare che due esami che hanno candidati in comune siano fissati nello stesso giorno. Supponiamo che il primo giorno di esami vogliamo mettere in calendario l'esame a. Dobbiamo adesso chiederci: quali esami possimo fissare nello stesso giorno? Certamente nessuno degli esami rappresentati da vertici adiacenti ad a, perché questi vertici rappresentano esami che possono essere sostenuti da studenti che sostengono l'esame a. Quindi cancelliamo dal grafo tutti i nodi adiacenti ad a e le relative connessioni. Quello che ottieniamo è il grafo in Fig. 8. Adesso so che posso sicuramente

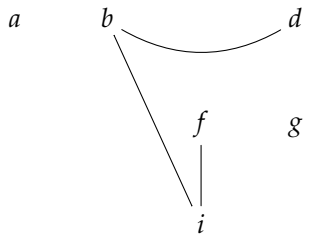


Figura 8: Grafo ottenuto dal grafo in Fig 7 cancellando tutti i nodi adiacenti ad a e le relative connessioni.

fissare per il primo giorno oltre all'esame a anche l'esame g. Ma posso anche fissare, nello stesso giorno, un esame fra b ed d, uno fra b e i e uno fra f e i. Per ottimizzare l'uso delle risorse disponibili (spazio, tempo, personale, etc.), mi conviene fissare nello stesso giorno anche altri esami "indipendenti". Decido di aggiungere l'esame b. Allora devo cancellare dal grafo, come prima, tutti gli esami connessi a b e le relative connessioni. Quello che ottengo è il grafo in Fig. 9. Quest'ultimo è costituito dall'insieme dei vertici a, b, f, g senza alcuna connessione fra loro (l'insieme E delle connessioni è vuoto). Un insieme di questo tipo si dice *indipendente*. I vertici rappresentano esami che possono essere fissati tutti al primo giorno. Si tratta anche di un insieme *indipedente massimale*, cioè non è possibile aggiungere nessun'altro esame nello stesso giorno senza violare il requisito secondo cui due esami che possono essere entrambi sostenuti da qualche

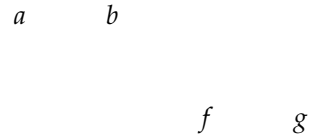


Figura 9: Grafo ottenuto dal grafo in Fig 8 cancellando tutti i nodi adiacenti a b e le relative connessioni.

studente non devono svolgersi contemporaneamente.

In questo modo abbiamo fissato il calendario per il primo giorno di esami. Allora torniamo al grafo in Fig. 7 e rimuoviamo tutti i nodi corrispondenti ad esami che abbiamo già fissati con le relative connessioni. Quello che otteniamo è il grafo in Fig. 10.

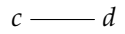


Figura 10: Grafo ottenuto dal grafo in Fig 9 cancellando i nodi a, b, f, g e le relative connessioni.

Adesso il nostro compito è già molto più semplice. Sappiamo che possiamo fissare nel secondo giorno gli esami e, h, i, dal momento che questi costituiscono un insieme indipendente di vertici. Ma questo insieme non è ancora massimale. Per renderlo massimale dobbiamo aggiungere un vertice fra c e d. Seguendo l'ordine alfabetico scegliamo c. Dunque l'insieme indipendente massimale che otteniamo è illustrato nella Fig. 11.

Figura 11: Grafo ottenuto dal grafo in Fig 10 cancellando il nodo d e la relativa connessione.

A questo punto, nel secondo giorno di esami, abbiamo fissato gli esami c, e, h, i. Dunque, togliendo dal grafo in Fig. 10 degli esami rimasti da collocare, anche gli esami che abbiamo appena fissato per il secondo giorno. Rimane soltanto il grafo costituito dall'unico nodo d. Anche questo è un insieme indipendente massimale (anche se un po' particolare). Concludiamo dunque il nostro compito fissando l'esame d nel terzo giorno di esami. In conclusione, il calendario degli esami che abbiamo ottenuto con questo metodo è il seguente:

Giorno 1	a, b, f, g
Giorno 2	c, e, h, i
Giorno 3	d

Questo esempio illustra bene il punto centrale di questo paragrafo. Se non avessimo organizzato la nostra informazione in modo appropriato, utilizzando un grafo per rappresentarla, il nostro compito sarebbe stato *molto* più difficile e probabilmente non avremmo trovato una soluzione altrettanto buona.

Ma quanto è buona questa soluzione? Siamo sicuri che non avremmo potuto organizzare gli esami in modo da svolgerli in due giorni invece che in tre? La nostra soluzione non scioglie questo dubbio. In fondo, per ottenerla, abbiamo fatto delle scelte. E se invece di fissare l'esame a il primo giorno avessi deciso di fissare uno qualsiasi degli altri? Non solo, anche lasciando inalterata la scelta di fissare a il primo giorno, una volta ottenuto il grafo in Fig. 8, abbiamo deciso di inserire l'esame b sempre il primo giorno, ma avremmo anche potuto scegliere d oppure i. Chi ci garantisce che facendo scelte diverse non avremmo ottenuto un risultato migliore, per esempio riducendo di un giorno il tempo da dedicare agli esami? Se il nostro obiettivo è quello di minimizzare il tempo da dedicare agli esami, non possiamo sapere se la nostra soluzione è *ottimale*. Per esserne sicuri, dovremmo ripetere il processo facendo scelte diverse fino ad esaurire tutte le possibilità, per poi confrontare i calendari così ottenuti. Ma, per far questo dobbiamo *enumerare* tutte le possibili soluzioni.

In questo caso specifico il compito è abbastanza complicato ma pur sempre fattibile. Dobbiamo ripetere il procedimento fino a esauirire tutte le possibili scelte. Già solo la scelta del primo esame che ci ha consentito di sfrondare il grafo passando dalla Fig. 7 alla Fig. 8 poteva essere fatta in 9 modi diversi. E per ciascuna di queste scelte diverse, ci sarebbe stata almeno un'altra scelta. Per esempio, la scelta che ci ha consentito di passare dal grafo in Fig. 8 a quello in Fig. 9 poteva essere fatta in 4 modi diversi. In generale, si tratta di un compito *estremamente* dispendioso in termini di tempo, soprattutto quando ci sono un gran numero di esami da mettere in calendario.

Questo tipo di problemi di *ottimizzazione*, problemi che non solo

ci chiedono di trovare una soluzione, ma di trovare anche la *miglior soluzione possibile* (rispetto a determinati obiettivi prefissati) sono di solito molto difficili da risolvere e, quando riguardano dati molto complessi, richiedono risorse computazionali eccessive persino per un computer di ultima generazione.

L'Algoritmo 1 illustra questa procedura utilizzando le istruzioni standard che abbiamo discusso prima.

Algorithm 1 Algoritmo per il calendario degli esami

1: **Input**: una lista ESAMI di esami e una relazione binaria R tra elementi di L tale che $R(x, y)$ se e solo se x e y hanno candidati in comune.
2: $L \leftarrow$ ESAMI
3: $E \leftarrow \{(x, y) \mid R(x, y)\}$
4: $D \leftarrow 0$
5: **while** $L \neq [\,]$ **do**
6: $D = D + 1$
7: **while** $E \neq \emptyset$ **do**
8: $H \leftarrow L$ ▷ *Introduce variabile H con valore iniziale L*
9: **repeat** ▷ *Cerca il primo elemento di L che è connesso ad altri vertici*
10: $z \leftarrow \mathsf{head}(H)$ ▷ *Sia z il primo elemento di H*
11: $H \leftarrow \mathsf{tail}(H)$ ▷ *Rimuove z da H*
12: **until** $(\exists x)(z, x) \in E$ ▷ *Finquando trova un z connesso*
13: **for all** $y \in L$ **do** ▷ *Rimuove vertici connessi a z e relative connessioni*
14: **if** $(z, y) \in E$ **then** ▷ *Se y è un vertice connesso a z*
15: $L \leftarrow L - [y]$ ▷ *Rimuove y*
16: $E \leftarrow E \setminus \{(z, y)\}$ ▷ *Rimuove connessione fra z e y*
17: **for all** $w \in L$ **do** ▷ *Rimuove connessioni di y con altri vertici*
18: **if** $(y, w) \in E$ **then** ▷ *Se w è connesso a y*
19: $E \leftarrow E \setminus \{(y, w) \mid (y, w) \in E\}$ ▷ *Rimuove connessione fra y e w*
20: **end if**
21: **end for**
22: **end if**
23: **end for**
24: **end while**
25: **write** "GIORNO D: L" ▷ *Scrive la lista degli esami assegnati al giorno D*
26: $L \leftarrow$ ESAMI $- L$ ▷ *Rimuove gli esami già assegnati dalla lista ESAMI*
27: $E \leftarrow \{(x, y) \mid x, y \in L\}$ ▷ *Rimuove le connessioni degli esami già assegnati*
28: **end while**

L'output di questo algoritmo viene prodotto dall'istruzione 6.3 man mano che si rende disponibile. L'output finale sarà dato da tutti gli output prodotti da questi istruzione giorno per giorno. Notate che in questo algoritmo, la sola scelta che facciamo è associata all'istruzione 1. A seconda di come ordiniamo gli esami in una lista, otteniamo

soluzioni diverse. Questo fatto può essere sfruttato nel caso in cui l'ordine in cui gli esami si devono svolgere sia rilevante, cioè quando per qualche motivo vogliamo che certi esami si svolgano prima di altri. Altrimenti, il numero di soluzioni diverse che possiamo ottenere sarà più o meno uguale al numero di possibili liste diverse che possiamo costruire a partire dagli elementi di V. Se V contiene n elementi, il numero di liste diverse che possiamo ottenere è uguale $n!$ (fattoriale di n) cioè al prodotto dei primi n numeri naturali. Al crescere di n questo numero diventa enorme. Pensate che, con appena 10 esami, il numero di liste diverse che possiamo usare nell'istruzione 1 è 3628800. E con 20 esami arriviamo alla cifra astronomica di 2432902008176640000!

Quindi se il nostro problema è quello di confrontare tutte le soluzioni possibili per scegliere quella *ottimale* (rispetto a certi obiettivi prefissati) ci troviamo in un'impasse. Anche se *in linea di principio* il nostro algoritmo ci consente di risolverlo (se avessimo a disposizione un tempo illimitato), *in pratica* quando il numero di esami è molto grande nessuno vivrà abbastanza a lungo da vedere la soluzione, nemmeno se usa il computer più veloce che la tecnologia ci mette a disposizione.

I limiti degli algoritmi (e dei computer)

Problemi algoritmicamente insolubili

Il concetto di macchina di Turing consente di rendere precisa la nozione di *algoritmo*. Come abbiamo visto nella dispensa precedente, dal punto di vista intuitivo un algoritmo non è altro che una procedura che può essere, in linea di principio, eseguita da una macchina.

La *tesi di Turing* è la tesi secondo cui la nozione di macchina di Turing cattura in modo esatto il concetto intuitivo di algoritmo, cioè: Tesi di Turing

> TESI DI TURING: Se un problema può essere risolto da una macchina (computer), allora può essere risolto da una macchina di Turing.

La tesi di Turing non è una tesi *dimostrabile*, è una *congettura*, simile a molte teorie delle scienze naturali. Queste teorie possono essere *confutate*, ma non possono essere *dimostrate*.

Una *teoria* è una proposizione *universale*, e.g.:

> Tutti i corvi sono neri

Come ha spiegato Karl Popper, una teoria come questa può essere *confutata*: sappiamo che è falsa quando incontriamo un corvo che non è nero. Ma non può essere *dimostrata*: non possiamo esaminare tutti i corvi possibili (passati, presenti e futuri) e verificare che sono tutti neri. Anche se tutti i corvi che abbiamo esaminato fino ad oggi sono risultati tutti neri, c'è sempre la possibilità che domani troviamo un corvo che non è nero.

Per questo le teorie delle scienze naturali non sono *verificabili*, ma sono *falsificabili*. In *matematica*, invece possiamo *dimostrare* che un'asserzione universale è vera, e.g.:

> Per tutti i triangoli, la somma degli angoli interni è uguale a 180°

Lo dimostriamo ragionando sul concetto di triangolo e su altri concetti *astratti*, definiti indipendentemente dall'esperienza.

La tesi di Turing è una proposizione *universale*

> Tutti i problemi che possono essere risolti da un computer possono essere risolti da una macchina di Turing.

Questo implica che le operazioni di *qualsiasi* computer possono essere simulate da quelle di una macchina di Turing. Non possiamo *verificare* questa teoria: dovremmo esaminare tutti i possibili computer (passati, presenti e futuri) Ma possiamo *falsificarla*, mostrando un computer che non può essere simulato da una macchina di Turing.

Fino ad oggi la tesi di Turing non è stata confutata e la quasi totalità degli scienziati ritiene che sia vera. La tesi di Turing viene usata principalmente per mostrare che un problema *non può essere risolto* da un computer. A questo scopo è sufficiente mostrare che il problema non può essere risolto da una macchina di Turing. Il ragionamento assume la forma seguente (è un esempio di *modus tollens*, vedi Dispensa n. 3 del primo modulo):

1	Se il problema Π può essere risolto da un computer, allora Π può essere risolto da una MT
2	Π non può essere risolto da una MT
	Π non può essere risolto da un computer

La premessa 1 è la *tesi di Turing*. La premessa 2 può essere *dimostrata matematicamente* ragionando sul problema Π (inteso come insieme di domande chiaramente definite) e sul concetto di macchina di Tu-

ring (che è un concetto astratto chiaramente definito, come quello di triangolo). In tal caso, la conclusione è vera se è vera la tesi di Turing

Un esempio di un problema che non è risolubile algoritmicamente, e dunque non può essere risolto in generale da un computer è il seguente:

Problemi algoritmicamente insolubili

PROBLEMA DELLE EQUAZIONI DIOFANTEE Un'equazione della forma:

$$P(x_1, \ldots, x_n) = 0$$

dove P è un *polinomio* di grado arbitrario con *coefficienti interi*, ammette soluzioni in termini di numeri naturali?

Si è dimostrato che non può esistere una macchina di Turing che risolve questo problema. Dunque, se la tesi di Turing è vera, questo problema non può essere risolto dai computer. Un altro problema insolubile a noi ormai familiare è il famoso *problema della decisione*.

PROBLEMA DELLA DECISIONE NEI LINGUAGGI QUANTIFICAZIONALI: Dato un insieme Γ di proposizioni e una proposizione P di un qualsiasi linguaggio quantificazionale L,

P è deducibile da Γ ?

Questo problema era stato proposto da David Hilbert (1862-1943) come uno dei ventitré fondamentali problemi aperti della matematica al congresso internazionale di matematica che si svolse a Parigi nel 1900. Grazie alle sue idee, Turing riuscì a dare un soluzione negativa — il problema non può essere risolto da una macchina di Turing e dunque (se la tesi di Turing è vera) da nessun computer — nel suo articolo *On Computable Numbers, with an Application to the Entscheidungsproblem*.[3] La stessa soluzione negativa venne data indipendentemente, e più o meno nello stesso periodo, da Alonzo Church (1903-1955) nel suo lavoro *An Unsolvable Problem of Elementary Number Theory*,[4] basandosi sul concetto di *funzione effettivamente calcolabile*.

[3] *Proceedings of the London Mathematical Society*, Series 2, **42** (1937), pp. 230–265.

[4] *American Journal of Mathematics*, **58** (1936), pp. 345–363.

Un altro celebre problema algoritmicamente insolubile è il cosiddetto *problema della terminazione*. A volte le istruzioni di un programma sono fatte in modo tale che, per certi input, il programma comincia ad eseguire il calcolo e *non si ferma mai*. Considerate, per esempio, le seguenti due istruzioni per una macchina di Turing

$$(0, -, 1, A, >)$$
$$(1, -, 0, B, >)$$

Se una macchina di Turing con queste istruzioni riceve come input una qualsiasi stringa di A e B, termina immediatamente senza eseguire alcuna operazione. Se invece riceve come input una "stringa vuota" di caratteri, continua a scrivere

$$ABABABABABABAB...$$

all'infinito, *senza fermarsi mai*. In questo caso è semplice verificare che, con questo tipo di input, questa macchina entra in un "loop infinito" e non fornirà mai una risposta. Quando un programma è molto complesso e consiste di un gran numero di istruzioni può essere molto difficile fare questa verifica. Sarebbe dunque molto utile se questa verifica potesse essere fatta automaticamente da un computer. Ci vorrebbe un algoritmo che, quando riceve come input un programma P scritto in un certo linguaggio di programmazione (per esempio il linguaggio delle macchine di Turing) e una certa stringa di simboli che codifica un dato input I per P, fornisce come output "SÌ" se P termina regolarmente con una risposta (positiva o negativa) quando riceve I come input, e "NO" quando invece P entra in un loop infinito e non fornisce mai una risposta.

PROBLEMA DELLA TERMINAZIONE: Dato un programma per computer P e un input I:

P si ferma quando riceve I come input?

Nell'articolo citato sopra, Turing mostrò che non può esistere una macchina di Turing che risolve questo problema e mostrò il collegamento fra la sua insolubilità e quella del problema della decisione nei linguaggi quantificazionali.

Problemi intrattabili

La teoria della *complessità computazionale* è un affinamento della tradizionale teoria della computabilità che tiene conto delle *risorse* (spazio e tempo) che sono consumate da un algoritmo. Concentriamo sul *tempo*. Se un problema Π può essere risolto da un algoritmo, ciascun esempio di questo problema è codificato da una stringa di simboli che codifica i dati forniti come input all'algoritmo. Se i dati sono costituiti da numeri, questi saranno codificati in un certo modo da una stringa di simboli che verrà data all'algoritmo come input. Questa stringa di simboli avrà una certa lunghezza n che dipende dal particolare esempio del problema Π che stiamo considerando e che può essere vista come una misura della *complessità* di questo particolare esempio: più lunga è la stringa necessaria a codificare l'input, maggiore è la complessità dell'esempio che corrisponde a quell'input.

Tempo di esecuzione di un algoritmo

Il *tempo di esecuzione* $T(n)$ di un algoritmo misura il numero *massimo* di passi che un algoritmo deve eseguire per un input di lunghezza n. In altri termini, è il tempo *entro il quale* l'algoritmo dà una risposta per un input di lunghezza n. Per esempio, se $T(n) = 3n^2 + 2n + 5$, questo vuol dire che per qualunque input di lunghezza n l'algoritmo esegue al massimo $3n^2 + 2n + 5$ istruzioni elementari. Quando la funzione $T(n)$, come in questo esempio, è rappresentata da un polinomio (di qualsiasi grado) si dice che l'algoritmo funziona in *tempo polinomiale*. Per molti problemi complessi gli algoritmi migliori che abbiamo a disposizione funzionano in *tempo esponenziale*, cioè il loro tempo di esecuzione è espresso da una funzione come $T(n) = 2^n + 4n + 27$.

La funzione $T(n)$ che rappresenta il tempo di esecuzione di un algoritmo ci dà importanti informazioni su come cresce il tempo che un computer impiega per rispondere a un determinato input al crescere della lunghezza dell'input (cioè della lunghezza della stringa di simboli che codifica l'input). Gli studiosi di complessità computazionale sono concordi nel sostenere che un algoritmo è *eseguibile in pratica* solo se funziona in *tempo polinomiale*. Questo ci consente di distinguere i *problemi trattabili*, quelli che possono essere *sempre* risolti in pratica da un computer, da quelli *intrattabili*, cioè quelli che, anche se esiste un algoritmo che li risolve, nei casi più complessi richiedono risorse che li rendono *praticamente insolubili* mediante un computer.

Problemi trattabili e intrattabili

PROBLEMI TRATTABILI E INTRATTABILI: Un problema Π è *trattabile* se esiste un algoritmo per risolverlo che funziona in *tempo polinomiale*. Altrimenti se *non* esiste un algoritmo per risolverlo che funziona in tempo polinomiale, il problema Π si dice *intrattabile*.

Per esempio, se l'algoritmo più efficiente possibile per risolvere Π funziona in tempo

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

cioè in un tempo *esponenziale*, allora il Π è un problema intrattabile.

Ci sono ottime ragioni per sostenere che se un algoritmo funziona in tempo esponenziale, non può essere eseguito in pratica nella maggior parte dei casi. Per capirlo basta guardare la Tabella 1. Nella prima colonna è indicato il tempo di esecuzione dell'algoritmo e nelle colonne successive il tempo massimo impiegato da una macchina che esegue dieci milioni di operazioni al secondo per risolvere un problema descritto da un input di lunghezza $10, 20, \ldots, 60$. Come potete facilmente constatare, nel caso degli algoritmi che funzionano in tempo esponenziale (ultime due righe della tabella), questo tempo cresce a una velocità impressionante. Anzi, anche per input di

T(n)	10	20	30	40	50	60
n	0,00001	0,00002	0,00003	0,00004	0,00005	0,00006
n^2	0,0001 secondi	0,0004 secondi	0,0009 secondi	0,0016 secondi	0,0025 secondi	0,0036 secondi
n^3	0,001 secondi	0,008 secondi	0,027 secondi	0,064 secondi	0,125 secondi	0,216 secondi
n^5	0,1 secondi	3,2 secondi	24,3 secondi	1,7 minuti	5,2 minuti	13 minuti
2^n	0,001 secondi	1 secondi	17,9 minuti	12,7 giorni	35,7 anni	366 secoli
3^n	0,059 secondi	58 minuti	6,5 anni	3855 secoli	2×10^8 secoli	1.3×10^{13} secoli

Tabella 1: Tempo impiegato da una macchina che esegue dieci milioni di operazioni in relazione al tempo di esecuzione dell'agoritmo impiegato. Tabella tratta da M. Garey e D.S. Johnson, *Computers and Intractability: a Guide to the Theory of NP-completeness*, W.H. Freeman & Co., New York 1975.

complessità limitata per i quali gli algoritmi che funzionano in tempo polinomiale forniscono risposte in tempi più che accettabili, il tempo richiesto da un algoritmo esponenziale diventa talmente grande che si può tranquillamente affermare che una macchina che lo esegue non darà "mai" una risposta (guardate il tempo richiesto nelle ultime due righe per input di lunghezza 50 e 60).

T(n)	Computer attuale	100 volte più veloce	1000 volte più veloce
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31{,}6N_2$
n^3	N_3	$4{,}64N_3$	$10N_3$
n^5	N_4	$2{,}5N_4$	$3{,}98N_4$
2^n	N_5	$N_5+6{,}64$	$N_5+9{,}97$
3^n	N_6	$N_6+4{,}19$	$N_6+6{,}29$

Tabella 2: Dimensioni massime di un input per il quale un computer fornisce una risposta entro un'ora. Tabella tratta da M. Garey e D.S. Johnson, *Computers and Intractability*, cit.

Dunque se un problema è intrattabile, cioè se non può essere risolto da un algoritmo polinomiale, non è possibile risolverlo in pratica mediante un algoritmo e cioè non è possibile risolverlo in pratica per mezzo di un computer (se la tesi di Turing è corretta). Si potrebbe pensare che questa classificazione dei problemi in trattabili e intrattabili rispecchi solo dei limiti tecnologici che possono essere facilmente superati. In fondo, un computer che esegue dieci milioni di opera-

zioni al secondo (la velocità usata come riferimento nella Tabella 1) poteva essere uno strumento tecnologicamente avanzato quarant'anni fa, ma certamente non oggi, in un epoca in cui anche il vostro laptop è in grado di eseguire diversi miliardi di operazioni al secondo. Tuttavia, la Tabella 2 mostra chiaramente che, quando un problema Π è "risolto" da un algoritmo esponenziale, anche grandi avanzamenti tecnologici incidono in misura trascurabile sulle dimensioni massime di un esempio di Π che può essere risolto entro un'ora di calcolo.

Come potete constatare, nel caso di algoritmi che funzionano in tempo esponenziale anche un grande miglioramento nelle prestazioni dei computer influisce in modo trascurabile sulla complessità degli esempi che possono essere risolti entro un'ora. Dunque, se un problema è intrattabile, resterà *praticamente insolubile* per gli esempi più complessi, indipendentemente da qualsiasi progresso tecnologico possa essere realizzato.

Purtroppo fra i problemi intrattabili, o probabilmente intrattabili, vi sono centinaia di problemi di fondamentale imporrtanza teorica e pratica. I seguenti sono famosi esempi di problemi algoritmicamente risolubili ma intrattabili:

Esempi di problemi intrattabili

- *Aritmetica di Pressburger*, cioè l'aritmetica con la sola opertazione di addizione (intrattabilità dimostrata da Fischer e Rabin nel 1974).
- *Aritmetica con la sola moltiplicazione* (Fischer e Rabin 1974)
- *Geometria euclidea elementare* (Fischer e Rabin 1974)
- *Teoria degli ordinamenti lineari* (Meyer 1975)

Vi è un insieme di problemi importanti per i quali nessuno è mai riuscito a trovare un algoritmo che li risolva che funzioni in tempo polinomiale, ma nessuno ha mai dimostrato che si tratta di problemi intrattabili, e cioè che un tale algoritmo non esiste. I problemi che appartengono a questo insieme sono equivalenti fra loro in questo senso preciso: si può dimostrare che se esiste un algoritmo polinomiale che ne risolve uno, questo può essere facilmente trasformto in un algoritmo polinomiale per risolverne qualunque altro. Questi problemi vengono detti *NP-completi*.

In teoria della complessità computazionale si indica con P la classe di tutti i problemi che possono essere risolti in tempo polinomiale da una macchina di Turing (e quindi da un qualsiasi computer). La classe NP, invece, è la classe di tutti i problemi che possono essere risolti in tempo polinomiale da una macchina di *Turing non-deterministica*, cioè da una macchina di Turing in cui, a ciascun passo, è possibile eseguire diverse istruzioni e bisogna scegliere quale eseguire. Il tempo impiegato da una macchina di Turing non deterministica per

P e NP

risolvere un problema viene calcolato considerando solo le "scelte migliori", cioè quelle che conducono alla soluzione nel tempo più breve possibile. Il fatto che un problema possa essere risolto in tempo polinomiale da una macchina di Turing non-deterministica non implica affatto che possa essere risolto in tempo polinomiale anche da una macchina di Turing deterministica, in cui non è possibile compiere delle scelte e a ogni passo può essere eseguita una sola istruzione. Dunque non è detto che $P = NP$ e, anzi, la congettura dominante è che $P \neq NP$ (anche se nessuno lo ha mai dimostrato in modo definitivo). Un problema *NP-completo* è un problema Π che fa parte della classe NP e al quale è possibile ridurre tutti i problemi in NP. Se Π è NP-completo qualunque esempio di un problema π' in NP può essere tradotto facilmente e velocemente in un esempio di Π. Così qualunque algoritmo che risolve in tempo polinomiale Π risolve in tempo polinomiale anche Π'. I problemi NP-completi sono tutti problemi computazionalmente *molto difficili* per i quali non si è mai trovato un algoritmo che li risolva in tempo polinomiale. Inoltre, se si trovasse un tale algoritmo che ne risolve uno, si potrebbe facilmente adattare in modo da risolvere tutti gli altri. Per questa ragione, i problemi NP-completi o sono tutti risolubili in tempo polinomiale, oppure nessuno di essi lo è, e gli studiosi di complessità computazionale sono concordi nel ritenere che nessuno lo sia. Dunque si tratta di problemi molti difficili che sono molto probabilmente intrattabili.

Problemi NP-completi

Un famoso esempio di un problema NP-completo è il seguente

IL PROBLEMA DEL COMMESSO VIAGGIATORE: dato un insieme di città m e supponendo sia nota la distanza fra ciascuna città e l'altra, è possibile visitarle tutte e tornare al punto di partenza coprendo una distanza complessiva minore di un certo valore dato D?

Un altro esempio, che ha a che vedere con il problema del calendario degli esami che abbiamo studiato nel paragrafo precedente, è il seguente:

INSIEMI INDIPENDENTI: dato un grafo G e un intero positivo K,

G contiene un insieme indipendente di vertici di dimensioni maggiori o uguali a K?

Fra i problemi NP-completi (e dunque molto probabilmente intrattabili) vi sono *centinaia* di problemi teorici e pratici della massima importanza nei campi più disparati, fra cui:

- Teoria dei grafi

- Progettazioni di reti
- Teoria degli insiemi
- Management di database
- Pianificazione delle attività e della produzione
- Teoria della programmazione
- Algebra e Teoria dei numeri
- Teoria dei giochi
- Teoria dei linguaggi
- Ottimizzazione dei programmi
- Logica

Nel 1971 il matematico canadese Stephen Cook ha dimostrato che tutti i problemi NP-completi sono equivalenti al seguente problema di logica booleana:

IL PROBLEMA DELLA SODDISFACIBILITÀ: data una proposizione P di un linguaggio booleano L, esiste un mondo possibile in cui P è vera?

dunque se c'è una soluzione efficiente per questo problema, ce n'è una per tutti. Il teorema di Cook mostra che la logica booleana è al centro di una rete di problemi computazionalmente difficili per cui molti problemi computazionali di grande importanza teorica e pratica possono essere reinterpretati come un problemi di logica booleana.