PREFAZIONE

Col termine *problem solving* si intende la disciplina che si occupa di trovare procedimenti generali per risolvere problemi e non di fare semplici calcoli; in particolare quindi essa non fa riferimento a tecniche specifiche di soluzione di problemi matematici, ma identifica un'area di attività e di ricerca alla quale contribuiscono diverse discipline, non solo matematica e informatica, ma anche filosofia, psicologia e pedagogia: **pensare, ragionare, fare ipotesi** possono, infatti, essere visti come forme generali di *problem solving*.

Questo argomento ha quindi una storia molto articolata e radici in discipline diverse; tuttavia con la comparsa del **computer** il *problem solving* ha acquisito una particolare caratterizzazione dovuta allo sviluppo e alla diffusione della «**programmazione**»; le linee guida illustrate nel seguito traggono gran parte della loro giustificazione dalla esperienza accumulata in questo ambito specifico dell'Informatica.

Per introdurre l'argomento, si può pensare che una situazione caratteristica di *problem solving* si presenta quando:

- viene dato un problema o un obiettivo da raggiungere e la soluzione non è immediatamente identificabile;
- la ricerca del procedimento risolutivo comporta una qualche difficoltà.

Si precisa innanzi tutto che, nel *problem solving*, dato un problema generico, l'obiettivo è quello di trovare un procedimento applicabile per trovarne una soluzione e non quello di ottenere la soluzione di una particolare istanza di quel problema; detto in altri termini, la soluzione non è il risultato prodotto dalla esecuzione di un **programma** per computer, ma è proprio il programma. Per questo motivo le abilità di *problem solving* sono simili a quelle per ottenere programmi per *computer* e, di conseguenza, la programmazione è un terreno sul quale è possibile svolgere efficaci sedute di allenamento per acquisire le abilità generali di *problem solving*. Questo allenamento può essere portato avanti fino trattare metodologie che consentono al computer stesso di acquisire (alcune specifiche) **abilità di problem solving**: l'area dell'Informatica che si occupa di questi argomenti (apprendimento automatico o *machine learning*) è l'Intelligenza Artificiale.

Alla fine di questo modulo viene illustrato un programma che, in determinati contesti, esibisce abilità di *problem solving*.

Un procedimento risolutivo, in generale, è costituito da un elenco di formule che devono essere applicate per trovare la soluzione di un problema; l'applicazione di ogni formula fa fare un passo avanti verso la soluzione del problema. Un procedimento per risolvere un problema è del tutto simile a un ragionamento che a partire da certe ipotesi iniziali consente di giustificare una affermazione finale; il procedimento e il ragionamento possono quindi essere pensati come un percorso che congiunge diversi punti o stati: lo stato di partenza è descritto dai dati iniziali o dalle ipotesi, lo stato d'arrivo è rappresentato dal risultato finale e gli stati intermedi rappresentano le situazioni alle quali si perviene applicando via via opportune formule o regole di deduzione. Per trovare questo percorso, alle volte è sufficiente avere un'idea brillante per scoprire un punto intermedio cruciale, trovato il quale, tutto il resto del percorso diventa ovvio; altre volte il percorso si ottiene svolgendo una sequenza sistematica di tentativi: il metodo illustrato in questo modulo si basa sul processo delle scomposizioni successive di un problema in sottoproblemi che utilizza entrambe le strategie.

Gli ingredienti fondamentali coinvolti in una situazione di *problem solving* sono quattro.

- Uno stato iniziale in cui, data una esigenza (e constatata una situazione problematica), viene formulato un problema (tecnico) da risolvere (e sono esplicitate tutte le premesse e i dati iniziali).
- Uno stato finale, raggiunto il quale si riconosce che il problema è stato risolto (ed è disponibile la soluzione).
- Un contesto, cioè un insieme di formule, azioni o regole di deduzione utilizzabili per passare da uno stato al successivo e che possono essere "assemblate" per costruire il procedimento risolutivo.
- L'ambiente operativo in cui rendere esplicite ed effettive le descrizioni degli stati iniziali e finali e delle regole da applicare.

Problemi possono nascere in molti contesti e in ogni disciplina: nonostante questa varietà, gli ingredienti sono sempre quelli sopra elencati. Ovviamente potranno cambiare il contesto e l'ambiente operativo: per individuare un esperimento cruciale per «falsificare» una teoria scientifica si dovranno usare strumenti concettuali diversi da quelli necessari per trovare una dimostrazione di un teorema; in ogni caso rimane decisivo l'ambiente operativo, cioè il linguaggio per rappresentare gli stati e le regole di transizione.

L’obiettivo del modulo è quello di illustrare con esempi e applicazioni alcune strategie che consentono di acquisire competenze generali per la soluzione di problemi. In particolare, il *problem solving* viene presentato come una disciplina che si occupa di

- individuare ed esplicitare procedimenti risolutivi efficienti e di
- descriverli e comunicarli in modo efficace a un esecutore

La metodologia adottata e le tecniche descritte in questo modulo sono del tutto generali e consentono di utilizzare le competenze acquisite in ogni contesto e quindi non solo quando l’interlocutore o l’esecutore è un computer.

Con queste premesse, l’argomento di questo modulo verrà svolto con riferimento a problemi riconducibili ad una qualche forma di elaborazione della informazione (e quindi nell’ambito delle discipline che fanno riferimento all’Informatica quali **Programmazione**, **Teoria degli Algoritmi** e **Intelligenza Artificiale**). I lettori interessati alla posizione del problema nella sua forma più generale possono consultare la letteratura relativa alle **Scienze Cognitive**.

Per stimolare il lettore e facilitare l’apprendimento degli argomenti proposti, nel testo sono inseriti esercizi e domande in itinere. Le domande in itinere sono nella forma di test a risposta multipla (ogni risposta è corredata da un commento); in caso di incertezza, il lettore è invitato a leggere i commenti anche delle risposte che siano ritenute plausibili o incerte. Gli esercizi non contengono indicazioni per la loro soluzione; il lettore è invitato a discuterne la soluzione con eventuali compagni di studio o direttamente con il tutor del corso.

IL PROBLEM SOLVING ALGORITMICO

La risolubilità dei problemi

Nell'ambito del problem solving, di un problema non è importante la sua difficoltà quanto la sua effettiva risolubilità. Questo argomento è stato a lungo dibattuto dai filosofi e dai matematici (Aristotele, Leibniz, Frege, Church e Turing, per citare solo i più noti) e negli anni trenta del secolo scorso è stata data la definizione della classe dei problemi effettivamente risolubili (in linea di principio); successivamente, con l'avvento dell'Informatica, è stato anche definito un criterio per riconoscere quando un problema è praticamente trattabile (con le risorse disponibili su un computer). Infatti, non tutti i problemi, nemmeno in linea di principio, possono essere risolti da un computer, comunque potenti e sofisticati siano l'hardware e il software disponibili. Per discutere questa affermazione, vengono introdotti i seguenti argomenti:

- Procedura effettiva e calcolo meccanico.
- Problema algoritmico.
- Algoritmo.
- Modelli (concettuali e operativi) di calcolo.
- Tesi di Church (definizione di calcolabilità).
- Esempi di problemi (o funzioni) non calcolabili.

PROCEDURA EFFETTIVA E CALCOLO MECCANICO

In questo contesto è fondamentale la definizione intuitiva di "procedura effettiva"; la effettività fa riferimento al fatto che l'attività di cui si parla deve essere eseguibile senza alcuna ambiguità. Per esempio possono essere considerati effettivi tutti i calcoli che prevedono l'esecuzione di (un numero finito di) operazioni di addizione e sottrazione fra numeri interi (poiché si ritiene scontato che le persone cui ci si rivolge le sappiano eseguire); inoltre, anche tutti i procedimenti riducibili a una lista finita di queste operazioni possono essere considerati effettivamente calcolabili.

Più in generale, se si dispone di un meccanismo che sa eseguire un certo numero di operazioni (non solo aritmetiche), si può ritenere effettiva qualsiasi procedura che preveda di eseguire solo (un numero finito di) queste operazioni; il procedimento viene eseguito in modo meccanico, quando sia indicata la prima operazione da eseguire e, dopo ogni operazione, sia individuata in modo esplicito e non ambiguo l'operazione che deve essere eseguita come successiva. Quindi un elenco ordinato di operazioni effettivamente eseguibili è un esempio di procedura eseguibile meccanicamente.

PROBLEMA ALGORITMICO.

Un problema viene definito algoritmico se valgono i seguenti due requisiti:

- è definito in modo effettivo l'insieme degli input legali per quel problema,
- è data una caratterizzazione effettiva dell' output come funzione dell'input.

Il problema può anche essere di tipo decisionale, cioè richiedere la verifica di una particolare proprietà, e l'output essere semplicemente **si** oppure **no**, a seconda che il particolare input soddisfi o no quella proprietà. I seguenti sono esempi di problemi algoritmici.

Problema 1. Dati i valori di base e altezza di un rettangolo (input), trovare l'area del rettangolo (output è il risultato di una moltiplicazione).

Problema 2. Dato un numero intero N (input), trovare il prodotto di tutti gli interi compresi fra 1 e N (output è il risultato delle successive moltiplicazioni).

Problema 3. Dato un insieme (finito) di numeri interi (input), disporli in ordine crescente (output è la permutazione ordinata).

Problema 4. Dati due insiemi (finiti) di parole (input), trovare l'insieme di quelle che sono presenti in entrambi (output è l'intersezione dei due insiemi dati).

Problema 5. Data una pianta stradale e due città presenti in essa (input), trovare il percorso più breve tra le due città (output è la lista ordinata di tratti stradali che costituiscono il percorso più breve).

Problema 6 (decisionale). Data una pianta stradale e un numero N espresso in chilometri (input), verificare se esiste un percorso che unisce tutte le città della mappa che sia di lunghezza minore di N (output è **si** oppure **no** a seconda che tale percorso esista oppure no).

Problema 7.(decisionale). Dato un numero intero positivo N (input), verificare se N è primo (output è **si** oppure **no** a seconda dell'esito della verifica).

Problema 8. (decisionale). Data una funzione Y = f(X) (X intero e Y razionale, per esempio Y = X/3) e un programma P con input un intero A e output un numero razionale B (input del problema è il programma P con il suo input e il suo output), verificare se per ogni intero A il risultato B fornito dal programma è uguale a f(A) (output è **si** se e solo se per ogni A si ha B = f(A)).

Attenzione!

Problema 9. Dato come input un testo scritto in una data lingua naturale (per esempio l'italiano), produrre come output il testo scritto in una diversa lingua naturale (per esempio l'inglese), con il vincolo che il secondo testo abbia ***il medesimo significato*** del primo.

Questo problema non è algoritmico. Infatti non è possibile in generale definire in modo esplicito e non ambiguo il significato di ogni frase corretta scritta in una lingua naturale e quindi non è possibile per ogni frase in input definire la corrispondente in output (avente il medesimo significato). (Se ci si limita esclusivamente alle frasi sintatticamente corrette con significato "effettivamente" non ambiguo, allora…).

ALGORITMO.

Per ogni problema algoritmico, ogni volta che si fissa un particolare input si definisce una particolare istanza di quel problema. Un problema (algoritmico) si può considerare risolubile (o computabile) se si esplicita un procedimento effettivo (eseguibile meccanicamente) che risolve qualsiasi istanza di quel problema (in un tempo finito). Un procedimento che abbia queste caratteristiche, cioè che risulti effettivo per ogni istanza del problema, si dice algoritmo.

E' ovvio che affinché un procedimento risulti effettivo è necessario che esso sia finito (non solo nel testo, ma anche nel tempo necessario per concludere il processo di calcolo).

Un problema per il quale esiste un algoritmo risolutivo si dice computabile.

In altri termini, l'algoritmo può anche essere definito come **la descrizione** di un procedimento che ha le seguenti proprietà:

- deve essere esplicita e non ambigua per l'interlocutore cui è destinata,
- il procedimento attivato di volta in volta seguendo il testo della descrizione deve terminare in un tempo finito, per ogni istanza del problema.

Il concetto di algoritmo presuppone quindi sempre la presenza di un linguaggio usato per la descrizione e di due interlocutori che lo sanno usare:

- chi produce la descrizione e

- chi sa eseguire tutte le azioni previste.

La parola algoritmo deriva dal nome del matematico arabo al Khuwarizmi vissuto nel secolo IX. Fino all'avvento del computer, gli algoritmi venivano usati essenzialmente solo in matematica: in questo caso il linguaggio era il gergo matematico e gli interlocutori erano due persone che lo sapevano usare per formulare l'algoritmo e per svolgere le operazioni descritte. Attualmente, il secondo interlocutore è quasi sempre un computer e il linguaggio usato per la descrizione del procedimento è un linguaggio di programmazione: in questo caso, l'algoritmo (cioè la descrizione del procedimento) assume la forma di un programma.

MODELLI (CONCETTUALI E OPERATIVI) DI CALCOLO.

Esistono problemi che sono risolubili con alcune macchine (che sanno eseguire un dato insieme di operazioni) e non lo sono con altre (che sanno eseguire altre operazioni)?

E' possibile che problemi, non risolubili con i computer di oggi, lo siano con quelli di domani? La questione della computabilità di un problema o calcolabilità di una funzione, come già detto, è stata dibattuta negli anni trenta del secolo scorso, prima della costruzione del primo computer e sono stati proposti diversi modelli di calcolo rispetto ai quali definire la computabilità (cioè la effettiva risolubilità) di un problema.

- Turing ha proposto un suo meccanismo concettuale (detto macchina di Turino) basato sulla capacità di scrivere e cancellare caratteri scritti su un nastro;
- Church ha proposto il così detto lamda-calcolo;
- Prost ha definito un meccanismo di manipolazione simbolica noto come regole di produzione;
- Kleene e Goedel hanno introdotto la classe delle funzioni ricorsive primitive.

Queste proposte hanno consentito di definire la effettiva computabilità o calcolabilità (relativa al meccanismo da loro proposto), cioè hanno trasformato la nozione intuitiva di calcolo effettivo in un concetto preciso; successivamente, tutti hanno dimostrato che con il loro meccanismo era possibile risolvere tutti i problemi per i quali si conoscevano soluzioni effettivamente calcolabili. Quindi, tutti questi meccanismi concettuali si sono dimostrati equipotenti, ciò che risultava computabile con uno di questi modelli risultava computabile anche con gli altri: tutti consentono quindi di definire una medesima classe di problemi computabili.

TESI DI CHURCH (DEFINIZIONI DI CALCOLABILITÀ).

Poiché tutti i modelli di calcolo (finora proposti) sono risultati equivalenti, Church ha proposto la seguente congettura (nota come tesi di Church)

UN PROBLEMA (O UNA FUNZIONE) E' COMPUTABILE, IN LINEA DI PRINCIPIO, SE E SOLO SE E' COMPUTABILE CON UNO DEI MODELLI DI CALCOLO SOPRA ELENCATI.

Poiché i linguaggi di programmazione dei computer sono modelli di calcolo equipotenti rispetto alla macchina di Turing, secondo Church non è possibile che la classe delle funzioni calcolabili (o dei problemi risolubili con un programma per computer) possa cambiare con la disponibilità di computer sempre più potenti.

La tesi di Church può quindi essere riformulata nel modo seguente: un problema è risolubile in linea di principio, se e solo se esiste un procedimento effettivo descrivibile con un linguaggio di programmazione. Si precisa che ogni algoritmo può essere formulato come programma per computer, mentre non è vero che ogni programma per computer sia la descrizione di un algoritmo; infatti, come si vedrà nei capitoli successivi, è estremamente facile scrivere un programma che non termina!

ESEMPI DI PROBLEMI ALGORITMICI NON COMPUTABILI (O INDECIDIBILI).

Come già detto, una funzione per la quale esiste un algoritmo per calcolarne i valori si dice calcolabile (in caso contrario si dice non calcolabile); i problemi si dicono (rispettivamente) computabili o non computabili (oppure decidibili o indecidibili se sono problemi di decisione). Tutti i problemi algoritmici con input finiti sono computabili, cioè è possibile trovare per essi un algoritmo che li risolva. In questi casi si tratta di costruire una corrispondenza fra ogni elemento dell'insieme (finito) di input e il corrispondente output: l'impresa può risultare ardua, ma è possibile.

La questione è interessante per i problemi che ammettono un insieme infinito di input ammissibili; in questo caso infatti non è immediato stabilire se esiste un algoritmo, cioè una procedura finita che in un tempo finito sa risolvere una qualsiasi delle possibili istanze del problema.

Il più noto (e il più citato) problema algoritmico non calcolabile è il così detto problema della terminazione di un programma che consiste nel decidere se quel programma termina per ogni suo input ammissibile.

Altro problema non computabile è il così detto problema del ricoprimento. In questo problema sono assegnate mattonelle quadrate di diversi tipi, ogni tipo è caratterizzato dai colori presenti su ciascuno dei quattro lati; l'input è dato da un insieme finito di tipi di mattonelle. Il problema consiste nel decidere se con mattonelle appartenenti a un assegnato insieme di tipi è possibile ricoprire una porzione qualsiasi di pavimento, col vincolo che due mattonelle possono essere accostate solo se sul lato comune i colori combaciano.

Anche il problema algoritmico presentato in precedenza come problema (decisionale) numero 8 (decidere se un programma fa quello che ci si aspetta che faccia !) è indecidibile.

La indecidibilità di questi problemi è stata dimostrata; cioè è stato dimostrato che non è possibile trovare procedimenti effettivi che risolvano **qualsiasi** istanza di questi problemi.

Per ottenere queste dimostrazioni di decidibilità o indecidibilità è sufficiente mostrare che un problema è o non è computabile o decidibile con uno qualsiasi dei modelli sopra riportati. Il risultato (positivo o negativo) ottenuto usando uno qualunque dei modelli presentati (per esempio la macchina di Turing), secondo la tesi di Church, ha valore universale, vale cioè anche per qualsiasi altro modello o strumento di calcolo. Ovviamente il risultato della dimostrazione, secondo la tesi di Church, vale anche per qualsiasi calcolatore e qualsiasi linguaggio di programmazione presente e futuro.

Sulla decidibilità del problema 9 non è possibile fare alcuna affermazione perché non è un problema algoritmico (e quindi, in linea di principio, non è risolubile con procedimenti effettivi di elaborazione di informazione). In pratica, per la soluzione di questo problema sono disponibili solo approssimazioni (più o meno accettabili).

Esigenza, Problema e Programma

Esigenza-Problema

Nel mondo del lavoro e nella vita in genere, la posizione di un problema trae, in generale, origine dalla necessità di far fronte a una esigenza scientifica, tecnica, economica o sociale. Una delle condizioni necessarie affinché abbia senso cercare un procedimento di soluzione effettivo (e ricorrere all'utilizzo di un computer) per risolvere il problema è di formularlo in modo algoritmico, cioè mediante:

- la descrizione dell'insieme di tutti gli **input** ammissibili,
- la descrizione dell'**output** atteso per ciascuno degli input.

Per raggiungere questo obiettivo, scelto il linguaggio da usare per esplicitare tutte e descrizioni necessarie, si devono eseguire le seguenti tre attività:

- individuare dal testo del problema le entità coinvolte in input e in output,
- definire per ciascuna entità l'insieme da cui possono prendere i rispettivi valori,
- esplicitare la o le relazioni tra le entità di input e output.

Per esempio nell'ambito della matematica, il **problema generico** di trovare l'area di un rettangolo coinvolge due entità in input (le misure della base e dell'altezza) e una in output (la misura della superficie); queste misure possono essere assegnate con numeri razionali non negativi. La relazione fra input e output in questo caso è data dalla formula per calcolare l'area:

Area = Base × Altezza

Una **istanza specifica** di questo problema è la seguente: calcolare l'area del rettangolo con base 3 metri e altezza 2 metri.

La teoria della **Calcolabilità,** come abbiamo visto, definisce i criteri che consentono di dichiarare effettivamente risolubile un problema, cioè di definire la classe dei problemi per i quali esiste un algoritmo risolutivo. In questa teoria, viene reso preciso e formale il concetto intuitivo e informale di calcolo effettivo; un problema viene definito effettivamente risolubile se è possibile esibire (per esempio) una macchina di Turing (o un programma per *computer*) che lo risolva. In tal modo, si dimostra che esistono problemi formulati in modo algoritmico (cioè come trasformazione dell'informazione di *input* in informazione di *output*) che non possono essere risolti con procedimenti effettivi (descrivibili per esempio come macchina di Turing o come programma per

computer); in definitiva, non tutti i problemi posti in modo algoritmico hanno un algoritmo risolutivo!

Ricordato che l'**algoritmo** per risolvere un problema è un procedimento effettivo utilizzabile per risolvere qualsiasi istanza di quel problema, le strategie di *problem solving* di cui si parla in questo modulo possono essere applicate solo a problemi per i quali sia possibile individuare un algoritmo.

La teoria della **complessità** definisce una ulteriore condizione necessaria per applicare una strategia costruttiva per risolvere un problema: l'algoritmo individuato deve essere computazionalmente trattabile. Quando il tempo di soluzione di un problema dipende da un parametro n, detto anche dimensione del problema generico (ciò che si verifica per esempio quando l'input è un insieme di n elementi), può accadere che il numero di operazioni da eseguire per trovare l'output corrispondente (cioè la soluzione) mostri una crescita (rispetto ad *n*) come illustrato negli esempi seguenti:

- **lineare**: trovare il più grande di n numeri dati,
- **quadratica**: mettere in ordine crescente n numeri dati, (per questo problema esistono anche algoritmi "leggermente" più efficienti),
- **cubica**: risolvere un sistema di equazioni lineari, (anche per questo problema esistono algoritmi "leggermente" più efficienti),
- **esponenziale**: elencare i sottoinsiemi di un insieme di n elementi,
- **superesponenziale**: elencare tutte le permutazioni di n oggetti.

Con riferimento alla tabella che segue, si può affermare che se l'algoritmo disponibile per la soluzione del problema generico prevede una quantità di operazioni che cresce esponenzialmente con la dimensione dell'input, in generale quel problema non è trattabile, ma possono essere risolti solo casi associati a dimensioni estremamente limitate. Solo quando il numero delle operazioni cresce al più in modo polinomiale ha senso descrivere l'algoritmo con un linguaggio di programmazione e richiedere che un computer esegua il procedimento risolutivo (per qualsiasi istanza).

In definitiva, quindi, affinché abbia senso ricorrere a un computer per risolvere un problema, non solo il problema deve essere risolubile in linea di principio (cioè essere calcolabile), ma deve anche essere trattabile, cioè si deve conoscere per quel problema un procedimento risolutivo (cioè

un algoritmo) che richieda un tempo di esecuzione ragionevole (non esponenziale o superesponenziale).

Per comprendere il significato di «tempo di esecuzione ragionevole» si supponga che per risolvere un problema di dimensione n sia necessario un milionesimo di secondo per eseguire il «passo» elementare (cioè per elaborare il caso $n = 1$). Con l'utilizzo della tabella seguente, in cui sono riportati i valori delle funzioni polinomiali (con esponente 2 e 3), dell'esponenziale (2^n) e del fattoriale ($n!$), è facile rendersi conto che non è ragionevole affrontare la soluzione di un problema se non si dispone di algoritmi con complessità polinomiale

n	n^2	n^3	2^n	n!
1	1	1	2	1
10	$1{,}0\times10^2$	$1{,}0\times10^3$	$1{,}0\times10^3$	$3{,}6\times10^6$
20	$4{,}0\times10^2$	$8{,}0\times10^3$	$1{,}0\times10^6$	$2{,}4\times10^{18}$
30	$9{,}0\times10^2$	$2{,}7\times10^4$	$1{,}0\times10^9$	$2{,}6\times10^{32}$
40	$1{,}6\times10^3$	$6{,}4\times10^4$	$1{,}0\times10^{12}$	$8{,}2\times10^{47}$
50	$2{,}5\times10^3$	$1{,}25\times10^5$	$1{,}1\times10^{15}$	$3{,}0\times10^{64}$

La tabella divisa in colonne, nella prima colonna si trova il valore di n, mentre nelle altre colonne sono presenti i valori delle funzioni polinomiali con esponenti 2 e 3, dell'esponenziale 2^n e di n!

Infatti, con n minore di 10, il problema viene risolto in tutti i casi in pochi secondi; per $n = 30$ occorrerebbero circa 20 minuti con un algoritmo esponenziale e non basterebbe l'età dell'universo con un algoritmo fattoriale! Con gli algoritmi polinomiali il tempo rimane inferiore al secondo anche per $n = 50$. (Si ricorda che la definizione formale del fattoriale di un numero intero non negativo è la seguente:

per $n = 0$, $n! = 1$ e per $n > 0$, $n! = n \times (n\text{-}1)!$

Si deduce facilmente che il fattoriale di n è uguale al prodotto di tutti i numeri interi compresi fra 1 e n; quindi, per esempio, si ha $3! = 6$ e $7! = 5040$, da cui $50!$ Circa uguale a $3{,}0\times10^{64}$).

Con riferimento agli esempi richiamati all'inizio del paragrafo, la tabella mette chiaramente in evidenza come cresce il tempo per ottenere la soluzione di ciascuno di questi problemi in funzione della sua dimensione.

- Per trovare il più grande di n numeri dati, il tempo cresce linearmente (vedi prima colonna): per $n = 50$ il tempo è 5 volte superiore a quello per $n = 10$.

- Per mettere in ordine crescente n numeri dati, il tempo cresce in maniera (quasi) quadratica (vedi seconda colonna): per $n = 50$ il tempo è 25 volte superiore a quello per $n = 10$.
- Per risolvere un sistema di equazioni lineari, il tempo cresce in maniera (quasi) cubica (vedi terza colonna): per $n = 50$ il tempo è 125 volte superiore a quello per $n = 10$.
- Per elencare i sottoinsiemi di un insieme di n elementi, il tempo cresce in modo esponenziale (vedi quarta colonna): per $n = 50$ il tempo è circa 10^{12} volte superiore a quello per $n = 10$.
- Per elencare tutte le permutazioni di n oggetti, il tempo cresce in modo superesponenziale (vedi quinta colonna): per $n = 50$ il tempo è circa 10^{58} volte superiore a quello per $n = 10$.

Quanto detto su calcolabilità e complessità dei problemi trattati in questo modulo si può riassumere nel modo seguente:

- il problema deve essere formulato in modo algoritmico (specificando le entità di input quelle di output e la relazione che le lega);
- deve esistere un algoritmo per la soluzione di tutte le istanze ammissibili di quel problema;
- il numero di operazioni da eseguire per ottenere la soluzione delle singole istanze non deve crescere in modo esponenziale o superesponenziale con il crescere della dimensione dell'input (e quindi non è sufficiente esibire un algoritmo e dimostrare che il procedimento ha termine).

Una volta trovato il procedimento effettivo e ottenuti i risultati, si devono sempre verificare due cose:

- che il procedimento sia effettivamente corretto rispetto al problema (algoritmico) che si sta trattando;
- che la soluzione del problema (algoritmico) sia anche una soddisfacente soluzione delle esigenze che lo hanno prodotto.

La prima di queste questioni è tecnica e viene affrontata nell'ambito della disciplina chiamata **correttezza** dei programmi; la seconda è di natura più ampia e coinvolge le discipline entro le quali hanno avuto origine le esigenze in oggetto; spesso questo argomento riguarda aspetti non solo puramente scientifici, ma anche sociali, etici o economici.

Linguaggio naturale, (pseudo)linguaggio e linguaggio di programmazione

In questo modulo, come già detto, col termine *problem solving* si fa riferimento all'insieme di competenze/abilità necessarie per portare a termine con successo la soluzione di un problema (generico), cioè quelle abilità che consentono di individuare un algoritmo trattabile e quindi un procedimento programmabile per un computer.

Poiché non esiste, a sua volta, un procedimento effettivo per affrontare la questione nella sua generalità, ci si deve limitare a dare alcune linee guida che si sono dimostrate efficaci nella pratica. Le linee guida che vengono illustrate hanno mostrato la loro validità nella esperienza di **programmazione**. Il *problem solving* può quindi essere pensato come una disciplina che si occupa di individuare gli algoritmi risolutivi (efficienti) e di descriverli in modo utile per l'esecutore. In Informatica, questo argomento è condiviso con la "Teoria degli algoritmi" e con la "Programmazione".

Per raggiungere questo obiettivo, saranno introdotti:

- il metodo delle scomposizioni successive di un problema (complesso) in sottoproblemi (via via più semplici), come strumento metodologico per individuare procedimenti di soluzione effettivi;
- uno (pseudo) linguaggio di programmazione, come strumento tecnico per descrivere formalmente in modo esplicito i problemi (semplici o complessi) e la loro (eventuale) scomposizione in sottoproblemi.

Nel contesto del *problem solving*, sono generalmente presenti tre ruoli:

- chi ha (l'istanza di) un problema,
- l'interlocutore che deve formalizzare il problema generico e trovare un procedimento effettivo per risolverlo,
- l'esecutore che deve fornire le soluzioni alle singole istanze del problema generico.

Nelle diverse situazioni questi ruoli possono essere rivestiti da uno, due o tre «attori» diversi.

In ogni caso, un problema può essere portato a soluzione solo se il primo «attore» riesce a comunicare al secondo un testo che descrive il problema generico e il secondo riesce a comunicare

al terzo il procedimento (effettivo) di soluzione; in questa prospettiva sono centrali il ruolo e la potenzialità espressiva del linguaggio che viene scelto per formalizzare la comunicazione e che gli attori devono condividere (almeno a due a due).

La centralità del linguaggio non cambia se chi ha il problema, chi lo formalizza e chi esegue il procedimento sono la stessa persona!

Vale la pena di ricordare che il computer può svolgere completamente il terzo ruolo, mentre è controverso (Intelligenza Artificiale) il suo coinvolgimento nei primi due.

In precedenza è stato detto che il problema generale di produrre la traduzione di un testo scritto in una lingua naturale non è calcolabile (cioè la traduzione non è eseguibile, in generale, usando un programma) perché non è possibile definire il significato di quel testo. Invece, il problema di tradurre un testo scritto in un linguaggio di programmazione è calcolabile (perché un testo corretto scritto in un linguaggio di programmazione ha uno e un solo significato). I programmi che eseguono le traduzioni fra testi scritti in linguaggi di programmazione si dicono interpreti o compilatori (le differenze tecniche fra i due tipi non rientrano negli obiettivi di questo modulo). Un programma traduttore (interprete o compilatore) è caratterizzato da tre linguaggi:

- il linguaggio usato per scrivere il messaggio che deve essere tradotto (detto sorgente),
- il linguaggio da usare per fornire la traduzione (detto oggetto),
- il linguaggio in cui è scritto il programma che effettua la traduzione.

I computer quando escono dalla catena di montaggio sanno usare un loro linguaggio specifico detto linguaggio macchina. Utilizzando questo linguaggio macchina è possibile scrivere un programma (interprete o compilatore) per tradurre programmi scritti in un qualunque linguaggio di programmazione sorgente (o di alto livello); se il linguaggio oggetto è il linguaggio macchina di quel computer, si è ottenuto il risultato di poter eseguire su quel computer anche programmi scritti in un diverso linguaggio di programmazione (il linguaggio sorgente). Sfruttando questa possibilità, si sono imposte nel tempo due famiglie di linguaggi di programmazione:

- a basso livello (o linguaggi macchina) per facilitare e rendere economica la costruzione dei computer,
- ad alto livello (linguaggi usati dalle persone) per facilitare la scrittura di algoritmi nelle diverse aree disciplinari e rendere efficienti i processi di elaborazione.

Per gli argomenti connessi con la disciplina del problem solving sono rilevanti solo i linguaggi di alto livello; questi, in relazione alle loro caratteristiche generali, possono essere classificati in 4 categorie: linguaggi procedurali, linguaggi dichiarativi, linguaggi funzionali e linguaggi ad oggetti. Dal punto di vista applicativo, sono utilizzati programmi (interpreti o compilatori) che traducono testi scritti in linguaggi (sorgente) di alto livello in testi scritti in linguaggi (oggetto) di basso livello; in linea di principio, possono esistere traduttori anche fra linguaggi del medesimo livello.

Per la scelta del linguaggio di programmazione da utilizzare nei processi di *problem solving* sono possibili due alternative significative che hanno rilevanza nella determinazione delle linee guida per disciplinare il processo:

- linguaggio procedurale: consente di fare riferimento a un insieme di azioni elementari (primitive) che l'esecutore sa eseguire;
- linguaggio dichiarativo: consente di fare riferimento a un insieme di problemi elementari (primitivi) che l'esecutore sa risolvere.

In pratica, questa scelta di natura «filosofica» si riconduce alla scelta del tipo di linguaggio di programmazione che si assume come riferimento. Se si assume di usare un linguaggio logico-dichiarativo (come il Prolog), si adotta la «filosofia» dichiarativa e le linee guida tendono a rispecchiare la struttura del problema da risolvere (il programma cui si perviene rispecchia la logica del problema più che quella del procedimento risolutivo). La scelta della «filosofia» dichiarativa adottata in questo modulo è dettata dalla evidenza sperimentale che sembra indicare che la competenza di *problem solving* acquisita per questa via abbia un carattere più generale e sia più facilmente trasferibile nei vari contesti disciplinari.

La differenza fra i due punti di vista non si apprezza su problemi semplici (come per esempio quelli che si risolvono applicando una semplice formula aritmetica); d'altra parte per descrivere la soluzione di problemi anche di modesta complessità, per i quali i due punti di vista sono significativamente diversi, occorrono sia nozioni che sono esposte successivamente in questo modulo sia quelle esposte in un modulo analogo sulla programmazione procedurale. Il lettore potrà, di volta in volta, decidere quale punto di vista adottare, in relazione alle sue particolari inclinazioni e (soprattutto) ai contesti concreti in cui i problemi si pongono.

In ogni caso l'abilità di *problem solving* (così come quella di programmazione) può essere sempre pensata come una sorta di abilità linguistica che si esprime mediante l'utilizzo di (pseudo)linguaggi di programmazione.

Come ricordato, la principale differenza fra linguaggio naturale e **linguaggio di programmazione** consiste nel fatto che le frasi scritte in un linguaggio di programmazione, se corrette, hanno sempre uno e un solo significato (che dipende dalla struttura sintattica della frase, la qual cosa consente di eseguire automaticamente la traduzione di testi scritti in linguaggi di programmazione), mentre questo in generale non è vero per le frasi (corrette) scritte in un linguaggio naturale (il che rende in linea di principio impossibile la traduzione di testi scritti in linguaggi naturali).

In alcune situazioni, per facilitare l'apprendimento della programmazione (cioè rendere più semplice il passaggio dalla espressione linguistica naturale a quella artificiale) si introducono pseudo linguaggi di programmazione che hanno una forma molto simile a quella del linguaggio naturale con regole di composizione sufficientemente disciplinate da somigliare a un linguaggio di programmazione; quindi un primo passo per acquisire le abilità che caratterizzano la programmazione è di abituarsi ad un uso disciplinato della lingua naturale, utilizzando costrutti linguistici formali privi di ambiguità.

I METODI TOP DOWN E BOTTOM UP

I problemi e le azioni primitive

I problemi primitivi sono quelli che l'interlocutore sa risolvere direttamente; in pratica è possibile "istruire" l'interlocutore in modo semplice e diretto su come risolvere in generale questi

problemi. Per ottenere poi la soluzione di una specifica istanza di un problema primitivo (da un computer o da qualsiasi altro interlocutore) si richiede solo di specificare l'istanza utilizzando il linguaggio condiviso. I problemi primitivi sono elementi tipici dei linguaggi dichiarativi e saranno ampiamente utilizzati in questo modulo.

Le azioni primitive sono quelle che si sanno descrivere con frasi elementari del linguaggio (che, per esempio, un interlocutore umano sa eseguire a memoria senza ausili tecnici). Per ottenere l'esecuzione di azioni primitive (da un computer o da un qualsiasi altro interlocutore) si richiede di specificare tutte le informazioni necessarie per eseguirle. Le azioni primitive sono elementi tipici dei linguaggi procedurali che non vengono utilizzati in questo modulo.

Sebbene le azioni primitive siano il corrispondente operativo dei problemi primitivi, l'utilizzo delle une o degli altri cambia radicalmente la metodologia e le tecniche per arrivare alla descrizione di procedimenti risolutivi (di problemi complessi non primitivi).

Il metodo *top down*

Il metodo *top down* per trovare un procedimento di soluzione di un problema consiste nel partire dal problema stesso e nel riconoscere se questo è primitivo: in questo caso la soluzione sarà immediata. In caso contrario, il metodo consiste nell'individuare due o più problemi più semplici la cui soluzione equivale a trovare la soluzione al problema iniziale. Questo processo si dice di scomposizione del problema (iniziale) in sottoproblemi. Se uno o più prodotti della scomposizione non sono problemi primitivi, il processo di scomposizione viene iterato finché si ottengono solo problemi primitivi. Il linguaggio (dichiarativo) scelto deve poter descrivere ogni passo di questo processo e il programma è rappresentato dalla descrizione finale delle scomposizioni.

Il metodo *bottom up*

Il metodo *bottom up* per trovare un procedimento di soluzione di un problema consiste nel partire dalla conoscenza dei problemi primitivi che l'esecutore sa risolvere e nel riconoscere se la soluzione di una loro (semplice) sequenza è sufficiente per risolvere il problema dato. In caso contrario si deve trovare (per aggregazioni e giustapposizioni successive) una sequenza appropriata (spesso molto complessa) di problemi primitivi la cui soluzione equivale alla soluzione del problema dato. In questo caso il programma è rappresentato dalla descrizione della sequenza (complessa) di problemi primitivi da risolvere.

La costruzione (dal basso) dei programmi assomiglia un po' a un gioco combinatorio in cui occorre assemblare molti pezzi, scelti da un insieme molto numeroso, per trovare una configurazione (la composizione del programma) che soddisfi a condizioni assegnate. È intuitivo (e anche sperimentalmente dimostrato) che questo modo di procedere soffre di due difetti:

- esistono più configurazioni che possono soddisfare i vincoli espliciti assegnati; alcune di queste hanno caratteristiche indesiderate (come per esempio inutili complessità aggiunte)
- persone diverse ottengono soluzioni che soddisfano comunque i vincoli, ma sono molto diverse.

Per contro, il metodo *top down*, disciplinando il processo di scomposizione del problema, favorisce una maggiore semplicità e uniformità del risultato (cioè della scomposizione finale del problema che, di fatto, diventa la descrizione dell'algoritmo per risolverlo).

Si sottolinea il fatto che, come regola generale, è opportuno usare il metodo *top down* e ricorrere al metodo *bottom up* solo in casi tecnicamente molto particolari (che interessano solo gli specialisti della programmazione).

La generalizzazione del problema

La generalizzazione del problema consiste nel passaggio da una istanza specifica del problema alla sua formulazione generica (vedi problema algoritmico). In questo modulo, il significato del termine “generico” è quello tecnico spiegato nel testo e si riferisce unicamente a un problema formulato nella sua accezione più generale.

Come è già stato detto, calcolare l’area di un rettangolo che ha base uguale a 4 cm e altezza 5 cm non è propriamente un problema, ma una istanza particolare di problema; la sua formulazione generica è: calcolare l’area di un rettangolo conoscendo la base e l’altezza. Sapendo risolvere il problema in questa seconda formulazione (cioè nella formulazione generica), si sa risolvere qualsiasi istanza di quel problema: cioè, qualunque siano le misure della base e dell’altezza, si è in grado di calcolare l’area. Su un esempio così semplice di problema, la questione della generalizzazione sembra capziosa; nei problemi più complessi, trovare una generalizzazione appropriata del problema non è sempre facile e comunque la successiva fase di soluzione ne è sempre condizionata. In ogni caso, il procedimento risolutivo deve sempre essere costruito per il problema generico ed essere quindi valido per tutte le istanze ammissibili del problema (e non solo per alcuni casi particolari o istanze specifiche).

Come esempio di generalizzazione si consideri il seguente problema: dato un insieme di numeri relativi (cioè negativi, positivi e lo zero), estrarre gli elementi maggiori di zero. (Questo problema non è formulato in modo generico perché nel testo compare una costante, lo zero).

Una prima facile generalizzazione di questo problema si ottiene sostituendo lo zero con una quantità generica N, per cui la nuova formulazione diventa:

dato un insieme di numeri relativi, estrarre gli elementi maggiori di un dato numero N.

Questa seconda formulazione è valida qualunque sia il numero N preso a riferimento per realizzare l’estrazione e «contiene» quindi anche la precedente, perché è sufficiente porre in essa N uguale a zero.

Invece, il problema di estrarre da un insieme di numeri tutti quelli che sono maggiori di N_1 e minori di N_2 non è «contenuto» nella formulazione generale precedente. Una più adeguata generalizzazione di questi problemi di estrazione di elementi da un insieme può essere la seguente:

dato un insieme di elementi (quindi non solo numeri) estrarre tutti quelli che verificano una assegnata condizione.

LA SOLUZIONE DEI PROBLEMI PRIMITIVI

Descrizione del problema

Come già detto, nel contesto del *problem solving* sono individuabili tre ruoli: chi ha il problema, chi deve individuare e descrivere un procedimento effettivo per risolverlo e chi deve eseguire il procedimento per trovare la soluzione delle specifiche istanze. Il ruolo centrale è il secondo che comporta due abilità strettamente connesse:

- saper formulare tecnicamente un problema in forma generica e trovare un procedimento effettivo per risolverlo e
- saper descrivere in modo esplicito e non ambiguo i procedimenti effettivi.

Nei paragrafi precedenti è stato presentato il contesto generale (di interesse comune ai primi due ruoli) in cui è inserito l'argomento *problem solving*; in particolare sono stati introdotti i concetti di:

- problema algoritmico (consistente in una elaborazione di informazioni: assegnato un input, trovare un output),
- calcolo meccanico effettivo (la computabilità),
- la trattabilità dei problemi (la complessità computazionale) e
- il metodo *top-down* (ovvero la scomposizione del problema –complesso- da risolvere in sottoproblemi via via più semplici).

Ora viene illustrato con esempi specifici come effettuare la scomposizione dei problemi complessi, abilità questa di specifico interesse per il ruolo del *problem solver*. Per raggiungere questo obiettivo si inizia facendo uso del linguaggio naturale con l'introduzione di alcune (semplici) strutture formali necessarie per rendere esplicite e non ambigue le descrizioni; solo quando si dovrà procedere alla stesura del testo (del programma) da passare all'esecutore si farà uso di uno specifico linguaggio di programmazione.

Il processo si articola nei quattro passi seguenti:

- formalizzazione del problema,
- elencazione di alcuni casi di prova,
- scrittura del programma,
- verifica.

Il primo passo formale nella soluzione di un problema (la formalizzazione) è quello di individuare le entità in esso coinvolte (citate esplicitamente o implicitamente nel testo del problema), ovvero di formulare il problema come un problema algoritmico; questo passo ha come obiettivo immediato quello di:

- dare un nome al problema (che dovrà essere una **stringa** che consente di individuarlo in modo univoco);
- individuare le entità (**variabili**) coinvolte e i rispettivi insiemi in cui esse prendono valori;
- comprendere la **relazione** che esiste tra le entità individuate.

La struttura formale usata è il «**termine**» il cui uso viene illustrato con alcuni esempi.

Primo esempio di formalizzazione di un problema.

Problema: `calcolare il volume della sfera, conoscendo il suo raggio.`

In questo caso le entità sono il raggio della sfera e il suo volume (che possono assumere valori numerici non negativi); mentre la relazione tra le entità coinvolte nel problema è rappresentata dalla formula studiata in geometria ($V = 4 \cdot \pi \cdot r^3/3$). In questo caso, il termine con il quale viene formalizzato il problema è il seguente:

```
{volume-sfera(Raggio,Volume)}.
```

Si sottolinea che in questa formalizzazione è stata anche resa esplicita la scelta relativa all'ordine in cui elencare le entità scritte entro le parentesi (decisione desumibile dai nomi usati per indicare le entità).

Per scrivere il termine relativo alla formalizzazione di un problema, utilizzando la convenzione illustrata , si deve quindi:

- scegliere una stringa che inizia con una lettera minuscola dell'alfabeto, come nome del problema (in questo caso la stringa scelta è `volume-sfera`);
- dare un nome alle entità, scegliendo per ciascuna una stringa con lettera iniziale maiuscola (in questo caso sono state scelte le due stringhe `Raggio e Volume`);
- decidere l'ordine secondo il quale elencare le entità (tra parentesi tonde separate da virgole),
- scrivere il termine fra parentesi graffe.

Una volta fissata questa formalizzazione, il problema deve sempre essere citato rispettando le convenzioni fatte:

- il nome del problema deve sempre essere il medesimo
- le entità devono sempre essere elencate, fra parentesi tonde, nel medesimo ordine.

Altri esempi di formalizzazione di problemi.

Problema: `mettere in ordine crescente gli elementi (per esempio dei numeri interi) contenuti in un elenco.`

Il problema fa ovviamente riferimento a due elenchi: quello iniziale e quello finale che contiene i medesimi elementi disposti in ordine crescente; la relazione tra input e output è basata, in questo caso, sul naturale criterio di ordinamento dei numeri interi. La formalizzazione può quindi essere scritta come:

`{ordinamento(Lista-iniziale, Lista-ordinata)}.`

Problema: `dati due elenchi, costruire un terzo elenco contenente nell'ordine tutti gli elementi contenuti nel primo elenco seguiti da quelli contenuti del secondo.`

Il problema fa riferimento a tre elenchi, e la relazione impone che il terzo contenga ordinatamente tutti gli elementi del primo seguiti da quelli del secondo; quindi la formalizzazione può essere scritta nel modo seguente:

`{giustapposizione(Primo-elenco, Secondo-elenco, Terzo-elenco)}.`

Problema: `date due città, appartenenti a un reticolo stradale descritto con i tronchi che congiungono le città a due a due, trovare un percorso stradale che le congiunga`.

Il problema fa riferimento a due città che devono essere gli elementi estremi di un percorso; quindi il percorso può essere descritto da un elenco ordinato di città, ciascuna delle quali è raggiungibile dalla precedente usando uno dei tronchi stradali disponibili: la relazione tra le entità è il grafo stradale in cui compaiono le città e i tronchi stradali che le uniscono. La formalizzazione può essere scritta nel modo seguente

`{collegamento(Città-di-partenza, Città-destinazione, Percorso)}.`

Si ribadisce che, per convenzione, le stringhe usate per i nomi dei problemi iniziano con lettera minuscola, mentre quelle usate per identificare le entità coinvolte iniziano con una lettera maiuscola. La formalizzazione è il primo passo del processo di soluzione; questo, assieme alla elencazione dei casi di prova, può essere considerato l'anello di congiunzione fra la formulazione in linguaggio naturale del problema e quella in linguaggio di programmazione del programma. In questo modulo la formalizzazione viene espressa in (pseudo)linguaggio di programmazione, scrivendo il termine fra parentesi graffe; se si rispetta il suggerimento di usare stringhe che, lette in linguaggio naturale, evocano le entità designate, risulta evidente l'ordine in cui queste entità vengono elencate nella scrittura del termine.

I casi di prova

Il secondo passo consiste nell'individuare alcuni casi di prova, cioè alcune istanze del problema, e di descrivere esplicitamente i valori assunti dalle entità per quelle istanze, valori che devono rispettare le relazioni esistenti fra le entità individuate. Questo passo esplicita direttamente la soluzione di istanze dei problemi particolarmente significative (casi semplici o casi estremi) e serve così di verifica della formalizzazione e della scelta corretta e appropriata delle entità.

L'individuazione dei casi di prova, con la loro descrizione formale, rappresenta una (prima) verifica per la formalizzazione adottata e per la comprensione della relazione coinvolta nel problema: se esiste un caso di prova che non si riesce a descrivere, significa che la formalizzazione adottata non è appropriata o che la relazione fra input e output non è stata compresa correttamente.

Formalmente, un caso di prova (che è una istanza specifica) viene descritto con lo stesso termine usato per il problema generico, dove però al posto delle variabili sono stati sostituiti dei

valori effettivi. Ovviamente, i valori devono essere riportati nel medesimo ordine in cui sono elencate le rispettive entità.

Con riferimento ai problemi riportati come esempi di formalizzazione, vengono illustrati alcuni casi di prova significativi. I casi di prova sono evidenziati dai caratteri `?-` posti a sinistra del termine e, come la formalizzazione, sono racchiusi fra parentesi graffe.

```
{?- volume-sfera(0, 0)}

{?- volume-sfera(1, 4.1887893)}

{?- volume-sfera(2, 33.5103144)}

{?- ordinamento([1, 3, 5, 4, 2], [1, 2, 3, 4, 5])}

{?- ordinamento([4, 5, 5, 6],[4, 5, 5, 6])}

{?- ordinamento([ ], [ ])}

{?- ordinamento([4],[4])}

{?- giustapposizione([2, 5, 3],[9, 8],[2, 5, 3, 9, 8])}

{?- giustapposizione([q, w, e, r], [ ], [q, w, e, r])}

{?- giustapposizione([ ], [z, x, c], [z, x, c])}

{?- percorso(bo, rsm, [bo, fc, rn, rsm])}

{?- percorso(ve, bz, [ve, pd, vr, bz])}

{?- percorso(bo, to, [bo, mo, pr, pc, al, to])}
```

In quest'ultimo esempio, le città sono indicate dalle rispettive targhe automobilistiche. Come si vede, gli elenchi sono rappresentati dai loro elementi racchiusi tra parentesi quadre; questa struttura si chiama lista e una maniera più formale di descriverla sarà illustrata in paragrafi successivi: la lista `[4]` rappresenta l'elenco formato dal solo elemento `4` e `[ ]` rappresenta l'elenco «vuoto», cioè privo di elementi; la lista è la struttura sistematicamente usata in questo modulo per rappresentare un elenco di elementi.

Le parentesi graffe usate per descrivere la formalizzazione del problema e i casi di prova segnalano che queste scritture non sono dirette all'esecutore, ma svolgono comunque un ruolo fondamentale per arrivare alla scrittura del programma perché sono il primo passo verso una scrittura formale. Ricordando la metafora dei tre ruoli, la formalizzazione del problema e la elencazione di alcuni casi di prova sono le strutture che devono essere ottenute e essere condivise da

chi propone il problema e da chi deve trovare il procedimento risolutivo (ovvero deve eseguire la scomposizione in sottoproblemi).

La soluzione dei problemi primitivi con formule.

Con la formalizzazione e la individuazione dei casi di prova (significativi) si può ritenere terminata la fase di *comprensione del problema* (il dialogo fra il primo e il secondo attore, chi ha il problema e chi deve capire come risolverlo): resta ora da affrontare il terzo passo, cioè la scrittura del programma (un soliloquio del secondo attore che terminerà con la stesura di un testo da consegnare al terzo attore, l'esecutore).

Come già detto, vengono detti primitivi i problemi particolarmente semplici che possono essere risolti o consultando una tabella (come per esempio un orario ferroviario o un elenco telefonico) o applicando una (semplice) formula aritmetica (come per esempio calcolare l'area di un rettangolo). Per esempio, il problema sopra riportato relativo al calcolo del volume di una sfera è un problema primitivo; mentre non sono primitivi il problema dell'ordinamento, quello della giustapposizione e quello del percorso stradale. Infatti questi ultimi tre, in generale, non sono risolubili applicando una formula aritmetica e neppure è pensabile di scrivere una tabella che, per l'ordinamento, contenga tutti i possibili elenchi da ordinare e, per la giustapposizione, elencare tutte le terne di elenchi che soddisfano la corrispondente relazione.

I programmi per risolvere problemi primitivi consistono semplicemente o nella scrittura della tabella da consultare o nella scrittura della (semplice) formula da applicare. Nel caso sopra visto, relativo al volume della sfera, il programma si scrive nel modo seguente

```
volume-sfera(Raggio,Volume) :- Volume = 4×3.141592×Raggio**3/3.
```

Cioè si scrivono il termine usato per la formalizzazione del problema e la formula che rappresenta la relazione fra input e output separate dal simbolo ":−". Questa struttura si dice **regola.** Il termine a sinistra del segno :− si dice testa della regola; la parte a destra di quel segno si dice corpo della regola: nei problemi primitivi, il corpo della regola contiene la formula che descrive la relazione tra le entità. Quando la formula contiene una potenza, questa si scrive inserendo un doppio asterisco fra la base e l'esponente (per esempio $Raggio^3$ viene scritto come Raggio**3).

Si fa notare che le eventuali costanti (come 3.141592 che è un valore approssimato di π) che compaiono nella relazione (cioè nella formula) non devono comparire nella struttura usata per la formalizzazione perché non sono variabili, cioè non sono entità che possono assumere valori diversi quando si devono risolvere istanze diverse del medesimo problema. Altri esempi di regole per la soluzione di problemi primitivi con formule sono:

```
area-rettangolo(Base, Altezza, Area) :- Area = Base×Altezza.

area-cerchio(Raggio,Area) :- Area = Raggio**2×3.141592.
```

La comprensione di queste scritture è facilitata dalla particolare convenzione adottata che suggerisce di usare (per i nomi dei problemi e delle entità) stringhe che evocano il problema e le rispettive entità coinvolte.

Data l'importanza pratica di questa convenzione si ribadisce il suggerimento guida già dato, di identificare le entità coinvolte nella formalizzazione sempre con stringhe che, lette come parole del linguaggio naturale, ricordino o evochino le entità che queste stringhe rappresentano nella formalizzazione adottata.

Nella scrittura del programma non sono più usate le parentesi graffe per indicare che questa scrittura è diretta all'esecutore (e di fatto è scritta rispettando le regole del linguaggio di programmazione Prolog). Le regole, mediante un opportuno *editor*, vengono inserite nella base di conoscenza dell'interprete/esecutore.

In questo modulo, il linguaggio **Prolog** non viene descritto in modo formale; sono illustrate, mediante esempi, solo le norme strettamente necessarie per rendere eseguibili su computer gli esempi trattati in questo testo.

Esercizi di scrittura di programmi per risolvere problemi aritmetici primitivi.

- Calcolare l'area di un trapezio.
- Calcolare l'area di una corona circolare.
- Calcolare il montante di un capitale di 1000 euro dopo cinque anni all'interesse composto del 3,5 %.
- Calcolare il resto dopo aver acquistato 6 confezioni di marmellata a 3 euro cadauna, avendo in tasca 20,45 euro.

NB. Per ogni problema, determinare il problema generico, scrivere il termine che lo rappresenta (con particolare attenzione alle entità coinvolte), individuare per ciascuno almeno tre casi di prova e scrivere la regola che rappresenta il programma per risolverlo.

La soluzione delle singole istanze dei problemi (aritmetici primitivi) viene richiesta utilizzando la struttura già vista per i casi di prova senza l'impiego delle parentesi graffe. Nel termine che rappresenta la formalizzazione del problema, le stringhe corrispondenti alle entità di input devono essere sostituite dalle costanti che caratterizzano la particolare istanza da risolvere, mentre le entità di output che rappresentano il risultato rimangono variabili.

Così, la richiesta di risolvere i seguenti problemi:

- calcolare il volume della sfera di raggio 3,45;
- calcolare l'area del rettangolo con base 4 e altezza 12,123

può essere formulata con le seguenti scritture

```
?- volume-della-sfera(3.45, V).
```

```
?- area-rettangolo(4, 12.123, A).
```

Per rispondere ad ogni domanda (cioè per risolvere l'istanza di un problema), l'esecutore cerca nella sua base di conoscenza una regola la cui testa sia un termine uguale a quello usato nella domanda: se non lo trova la risposta alla domanda è semplicemente NO; se lo trova, viene fatta una copia della regola utilizzando i valori degli argomenti usati nella domanda. Nel caso in esame, vengono trovate le regole corrispondenti che sono ricopiate rispettivamente come

```
volume-sfera(3.45,V) :- V = 4×3.141592×3.45**3/3.
```

```
area-rettangolo(4, 12.123, A) :- A = 4×12.123.
```

Fatti i calcoli il sistema fornisce il risultato ottenuto. Nei due casi in esame, l'esecutore/interlocutore (che è capace di eseguire i calcoli descritti dalla formula, dopo avere sostituito le variabili con le corrispondenti costanti, come sopra descritto) risponderà rispettivamente

```
V = 170.0
```

```
A = 48.492
```

Quando si vuole risolvere una istanza di problema di cui sia già stato scritto il programma, si devono rispettare le seguenti convenzioni:

- usare il medesimo termine definito nella formalizzazione del problema, copiando esattamente il nome del problema;
- sostituire le stringhe che rappresentano le variabili di input con costanti rispettando l'ordine in cui sono state disposte le entità corrispondenti entro le parentesi tonde;
- rappresentare le variabili di output con stringhe rappresentative di variabili (il primo carattere deve essere una lettera maiuscola); queste stringhe possono essere anche diverse da quelle usate nella formalizzazione del problema, il loro significato è legato alla posizione da esse occupate.

Esercizi.

Descrivere due interazioni con l'esecutore per ciascuno degli esercizi proposti nel paragrafo precedente.

La soluzione dei problemi primitivi con tabelle.

Esempi di problemi primitivi risolubili mediante consultazione di una tabella sono, per esempio, quelli che prevedono l'uso di un elenco telefonico, la consultazione della propria agenda personale o la produzione di certificati anagrafici.

Per ottenere da un esecutore la soluzione di problemi primitivi numerici, devono essere conosciute le formule da applicare e, scritta la regola, editarla nella base della conoscenza del sistema; per i problemi primitivi risolubili consultando una tabella, devono essere conosciute le tabelle da consultare. Per conoscere le formule si ricorre di solito alla matematica; nel secondo caso è l'interessato che deve costruirsi la tabella necessaria, ricorrendo eventualmente a pubblicazioni opportune. Un esempio di tabella è quello sotto riportato relativo agli stati che (per primi) hanno adottato come moneta l'Euro

STATO	CAPITALE	ABITANTI	SUPERFICIE
Austria	Vienna	8	84
Belgio	Bruxelles	10	31
Eire	Dublino	4	70
Francia	Parigi	59	544
Germania	Berlino	82	357
Grecia	Atene	11	132

Italia	Roma	57	301
Lussemburgo	Lussemburgo	0,4	3
Olanda	Aia	16	42
Portogallo	Lisbona	10	92
Spagna	Madrid	40	506
Svezia	Stoccolma	9	450

In questa tabella, su ciascuna riga, sono riportati il nome dello stato, quello della sua capitale, il numero degli abitanti (in milioni) e la superficie in chilometri quadrati per 1000. (I dati riportati nella tabella sono approssimati).

Per illustrare il metodo di soluzione di problemi primitivi mediante uso di tabelle, si fa riferimento a una tabella contenente i dati presenti in una ipotetica agenda personale. Nella tabella supponiamo siano riportati i seguenti dati

- Nome della persona,
- Cognome della persona,
- Numero telefonico,
- Professione.

La tabella avrà quindi 4 colonne; inoltre, ogni tabella deve avere un nome che è bene venga scelto in modo da ricordarne il contenuto.

La descrizione «implicita» della tabella in oggetto, che è equivalente alla formalizzazione dei problemi generici che si possono risolvere consultando questa tabella, può essere formalizzata con un termine racchiuso tra parentesi graffe, come illustrato nel modo seguente

```
{agenda(Nome, Cognome, Telefono, Professione)}.
```

Il nome della tabella è una stringa che inizia con una lettera minuscola dell'alfabeto; le stringhe che designano le colonne (cioè le entità coinvolte) iniziano con una lettera maiuscola.

Anche la comprensione di queste scritture (come già visto per i semplici problemi aritmetici) è facilitata dalla particolare scelta fatta per le stringhe che rappresentano la tabella e le rispettive entità coinvolte; si ribadisce quindi l'utilità del suggerimento già più volte ricordato:

scegliere le stringhe in modo tale che, lette come parole del linguaggio naturale, ricordino/evochino le entità che queste stringhe rappresentano nella formalizzazione adottata. L'uso congiunto di questo suggerimento e della adozione della struttura del termine per rappresentare i problemi sono parte dello (pseudo)linguaggio di programmazione adottato in questa presentazione.

Altro esempio di tabella desumibile da informazioni di stato civile (qui riportata in una versione semplificata) può contenere per esempio i seguenti dati:

- Nome del nato
- Nome del padre
- Cognome del padre
- Nome della madre
- Cognome della madre

La formalizzazione implicita di questa tabella può essere rappresentata nel modo seguente

```
{nascita(Nome,Npadre,Cpadre,Nmadre,Cmadre)}.
```

Come già detto, la descrizione «implicita», rappresentata da un termine scritto tra parentesi graffe, serve per definire la struttura della tabella e per specificarne il nome e il contenuto delle singole colonne.

Per risolvere istanze specifiche dei singoli problemi, sono necessarie le descrizioni «esplicite» delle singole tabelle che si ottengono utilizzando il termine definito nella corrispondente descrizione implicita, scrivendo un termine per ciascuna riga della tabella; la descrizione esplicita della tabella rappresenta il programma. Un esempio di descrizione esplicita della tabella di nome agenda sopra introdotta è la seguente.

```
agenda(mario, rossi, 3334445551, avvocato).
agenda(maria, verdi, 3334445552, docente).
agenda(dario, bianchi, 3334445553, geometra).
agenda(ilaria, bianchi, 3334445554, giudice).
agenda(marco, rossi, 3334445555, farmacista).
agenda(lucia, bini, 3334445556, impiegata).
```

Si fa notare che nello scrivere la forma esplicita deve essere rispettata la convenzione fatta con la forma implicita di utilizzare 4 argomenti che nell'ordine devono essere il nome della persona, il cognome, i numero di telefono e la professione. La rappresentazione esplicita della tabella, inserita nella base di conoscenza del sistema, è una forma di programma (equivalente alla forma della regola, già vista per i problemi che si risolvono con una semplice formula).

Anche la soluzione delle singole istanze dei problemi (primitivi) risolubili mediante la consultazione di una tabella viene richiesta utilizzando il termine definito nella fase di formalizzazione; anche in questo caso, le stringhe che rappresentano le entità di input devono essere sostituite dalle costanti che definiscono la particolare istanza da risolvere.

Per le entità che non assumono valori numerici, i dati di input (cioè le costanti) vengono descritti da stringhe con lettera iniziale minuscola, utilizzando le medesime convenzioni definite con la scrittura della forma esplicita; mentre i dati di output (cioè le variabili), come in precedenza, sono rappresentate con una stringa con l'iniziale maiuscola.

Consideriamo i seguenti problemi:

- Qual è il numero di telefono dell'impiegata Lucia Bini
- Che professione svolge e qual è il numero di telefono di Dario Bianchi
- Qual è il nome del sig. Rossi che fa il farmacista

Non è difficile riconoscere che questi problemi si possono risolvere tutti consultando la tabella `agenda` sopra vista; quindi la formalizzazione del problema generico è

```
{agenda(Nome, Cognome, Telefono, Professione)}.
```

Le richieste di risolvere questi problemi vengono di conseguenza formulate rispettivamente con le seguenti scritture

```
?- agenda(lucia, bini, Tel,impiegata).
?- agenda(dario, bianchi, Tel, Professione).
?- agenda(Nome,  rossi, _ , farmacista).
```

L'esecutore/interlocutore, per rispondere ad ogni domanda, cerca nella sua base di conoscenza la tabella descritta con il medesimo termine utilizzato nella domanda. All'interno di questa tabella (nel caso in esame la tabella `agenda`), cerca una riga che contenga le medesime costanti contenute nella domanda: quindi, per ogni variabile contenuta nella domanda, trova la costante che, in quella riga, ne occupa la medesima posizione. In questo caso, le corrispondenze tra domanda e riga della tabella sono le seguenti

```
?- agenda(lucia, bini, Tel       , impiegata).
   agenda(lucia, bini, 3334445556, impiegata).
```

```
?- agenda(dario, bianchi, Tel       , Professione).
   agenda(dario, bianchi, 3334445553, geometra).

?- agenda(Nome , rossi, _          , farmacista).
   agenda(marco, rossi, 3334445555, farmacista).
```

Infine, il sistema mostra l'accoppiamento variabile costante così trovato; in questi esempi, le risposte del sistema alle tre domande sopra riportate sono quindi nell'ordine le seguenti

```
Tel = 3334445556
Tel = 3334445553, Professione = geometra
Nome = marco
```

Nel terzo esempio, il testo del problema non fa riferimento ad una entità che è invece prevista nella formalizzazione della tabella che si deve usare; quando ciò avviene (per una o più entità), la posizione corrispondente alle entità non citate nel testo della domanda (né come costanti né come variabili) deve essere occupata dal segno speciale _ (sottolineatura).

Il meccanismo tecnico con il quale l'esecutore (dichiarativo) risponde alle domande, cioè risolve i problemi, si chiama unificazione e viene illustrato in relazione a problemi che si risolvono consultando una tabella distinguendo due casi.

Primo caso: la domanda contiene solo costanti.

Il processo di unificazione consiste nel verificare che nella tabella ci sia un termine del tutto uguale a quello usato nella domanda: se tale termine esiste, la risposta è SÌ; se non esiste, la risposta è NO.

Secondo caso: la domanda contiene una o più variabili.

Il processo di unificazione consiste nel determinare se esiste un termine nella tabella che ha tutte le costanti uguali (ordinatamente) a quelle contenute nella domanda: la risposta produce l'elenco delle variabili ciascuna associata alla costante che sostituita ad essa nella domanda rende il termine presente nella domanda uguale alla riga della tabella.

Naturalmente, in questo secondo caso, sono possibili risposte multiple in quanto può accadere che il termine della domanda possa unificarsi con più di una riga della tabella.

L’ordine con cui vengono ottenute le risposte multiple dipende dall’ordine in cui si trovano le righe della tabella. In generale, come si vedrà in seguito, l’ordine delle risposte multiple dipende sempre da come è scritto il programma.

Esercizi

1. Consultando la tabella agenda, formulare le domande e prevedere le risposte ai seguenti problemi.

- Qual è il numero di telefono del giudice.
- Che professione svolge Maria Verdi.
- Qual è il nome dell’avvocato.
- Quali sono nome e cognome del docente.

2. Con riferimento alla tabella implicita definita nel testo per le informazioni di stato civile, ipotizzare una tabella esplicita e simulare la soluzione di alcune istanze di problemi.

3. Fornire le descrizioni implicita ed esplicita della tabella degli stati che hanno adottato per primi come moneta l’euro e formulare le domande per risolvere i seguenti problemi.

- Trovare abitanti e superficie dell’Irlanda.
- Trovare la capitale della Svezia.
- Trovare la capitale e gli abitanti del Lussemburgo.

Problemi decisionali con risposta SÌ/NO

Come si è visto, la soluzione dei problemi che prevedono l'esecuzione di un (semplice) calcolo aritmetico viene ottenuta perché l'esecutore/interlocutore sa:

- utilizzare le costanti poste nella domanda, inserendole in «modo appropriato» nella formula specificata nel programma;
- eseguire le operazioni specificate nella formula
- restituire il valore trovato.

In modo analogo, la soluzione dei problemi che prevedono la consultazione di una tabella viene ottenuta mediante il processo di unificazione che consiste nelle seguenti azioni:

- individuare, se esiste, la tabella che ha lo stesso nome del termine presente nella domanda;
- individuare, esaminando ordinatamente le righe di quella tabella a partire dalla prima, una riga le cui costanti coincidano con quelle presenti nella domanda;
- sostituire le variabili presenti nella domanda con le rispettive costanti presenti nella riga individuata;
- restituire per ogni variabile i valori costanti usati nella sostituzione precedente.

Può succedere che nella tabella non sia presente alcuna riga contenente i dati presenti nella domanda; in questo caso non è possibile ovviamente individuare la riga che contiene quei dati e quindi la risposta dell'esecutore alla richiesta di risolvere il problema è semplicemente NO: il tentativo di unificazione è fallito. Per esempio la domanda

```
?- agenda(giorgio, neri, Tel, _).
```

con riferimento alla tabella del paragrafo precedente, provoca la risposta NO.

Può anche succedere che nella domanda siano presenti solo costanti; in questo caso il significato del problema non è quello di trovare informazioni, ma quello di verificare se nella tabella è presente una data riga: in questi casi la risposta dell'esecutore sarà semplicemente un SÌ, se quella riga esiste, oppure un NO, se quella riga non esiste. Per esempio le domande

```
?- agenda(dario, bianchi, 3334445553, geometra).
```

```
?- agenda(marco, rossi, 2224446666, farmacista).
```

con riferimento alla medesima tabella, provocano rispettivamente le risposte SÌ e NO

Anche con problemi di tipo aritmetico è possibile che il sistema possa rispondere semplicemente con un SÌ oppure con un NO. Infatti, le risposte alle seguenti domande

```
?- area-rettangolo(2,4,5).
```

```
?- area-rettangolo(3,6,18).
```

sono rispettivamente NO e SÌ perché

- non è vero che l'area di un rettangolo che ha base 2 e altezza 4 è 5,
- è vero che l'area di un rettangolo che ha base 3 e altezza 6 è 18.

Esercizi

Usando la tabella agenda sopra riportata, formulare alcune domande con risposta SÌ e alcune con risposta NO.

Ricapitolando, quando la domanda viene posta con un termine che contiene solo costanti (in questo caso, il termine si dice ground), il problema (decisionale) consiste in una verifica (è vero che …?) e quindi la risposta è del tipo SÌ/NO.

Quando la domanda viene posta con un termine che contiene variabili, il problema consiste nel determinare quanti sono i valori possibili (le sostituzioni) per le variabili; come si vedrà nei paragrafi successivi, spesso sono possibili, per le variabili, più sostituzioni.

Problemi con risposta multipla

Supponiamo di integrare la versione della tabella agenda dei precedenti paragrafi con informazioni relative ad altre persone; sia la seguente la nuova versione

```
agenda(mario,rossi,3334445551,avvocato).
agenda(maria,verdi,3334445552,docente).
agenda(dario,bianchi,3334445553,geometra).
agenda(ilaria,bianchi,3334445554,giudice).
agenda(marco,rossi,3334445555,farmacista).
agenda(lucia,bini,3334445556,impiegata).
agenda(mario,rossini,3334446661,docente).
agenda(maria,verdi,3334446662,avvocato).
agenda(mario,bianchi,3334446663,geometra).
agenda(ilaria,bianchi,3334446664,farmacista).
agenda(mario,folli,3334446665,farmacista).
agenda(lucia,boni,3334446666,impiegata).
```

Con tabelle molto ampie, non è infrequente che un problema possa avere più soluzioni. Se si deve, per esempio, risolvere il problema:

trovare nome e cognome di un farmacista con relativo numero telefonico

si deve porre una domanda che usi la formalizzazione della tabella `agenda` (che, si ricorda, ha quattro argomenti che, nell'ordine, sono nome, cognome, telefono e professione); nel testo del problema sono presenti una costante (la professione di farmacista) e tre variabili (nome, cognome e telefono); nella corrispondente domanda da porre all'esecutore si devono mettere come argomenti la costante «farmacista» nel quarto posto e tre variabili nelle prime tre posizioni; l'interazione completa con l'esecutore è quindi la seguente

```
?- agenda(Nome, Cognome, Tel, farmacista).
Nome = marco, Cognome = rossi, Tel = 3334445555;
Nome = ilaria, Cognome = bianchi, Tel = 3334446664;
Nome = mario, Cognome = folli, Tel = 3334446665;
```

```
NO
```

Come si vede, tutte le possibili soluzioni si ottengo digitando il punto e virgola (;) dopo ciascuna risposta; questo carattere segnala all'esecutore di cercare una ulteriore unificazione (continuando a consultare la tabella a partire dalla riga successiva a quella appena utilizzata); quando non esiste più alcuna riga unificabile con la domanda, la risposta è NO.

Esercizi

1. Con riferimento alla tabella

```
{agenda(Nome, Cognome, Telefono, Professione)}
```

integrata, trovare altri problemi con risposte multiple e formulare le rispettive domande.

2. Con riferimento alla tabella

```
{nascita(Nome, Npadre, Cpadre, Nmadre, Cmadre}
```

costruire la corrispondente tabella esplicita sufficientemente ricca e determinare almeno tre problemi per ciascuna delle seguenti tipologie

- risposta SÌ,
- risposta NO,
- risposta singola e
- risposta multipla.

Nei paragrafi precedenti sono state illustrate le fasi di formalizzazione, elencazione dei casi di prova e programmazione; resta da illustrare la fase di verifica. Questa fase fa parte del così detto problema della dimostrazione di correttezza dei programmi; data la complicazione tecnica, in questo modulo non è possibile trattare l'argomento. Non si può comunque non segnalare il fatto che nella scrittura di un programma è sempre possibile che siano presenti degli errori (non solo di forma ma anche di «sostanza»); viene quindi dato il seguente suggerimento che deve essere interpretato come un obbligo:

scritto il programma, sottoporre sempre, a mo' di verifica, tutti i casi di prova precedentemente definiti.

La individuazione di casi di prova significativi svolge quindi un ruolo fondamentale nell'attività di *problem solving* sia nella fase preliminare di comprensione del problema, sia in

quella finale di assegnare una patente di qualità al programma! Tuttavia va ribadito che i casi di prova possono solo falsificare un programma e mai dimostrarne la correttezza.

SCOMPOSIZIONI SEQUENZIALI E SUCCESSIVE

Riscrittura del problema

In genere la riscrittura del problema serve per rendere più semplice l'interrogazione di tabelle complesse. Può avvenire che le entità coinvolte in un problema siano un sottoinsieme di quelle descritte in una tabella già descritta e utilizzata per altre applicazioni; in questi casi, non è necessario riscrivere (parte di) quella tabella, ma è sufficiente usare una regola con il nuovo problema a primo membro e la descrizione della tabella al secondo: di fatto, questa riscrittura dice al sistema (cioè all'interlocutore) dove devono essere prese le informazioni necessarie per la soluzione del problema in esame. Gli esempi che seguono illustrano una applicazione di questa tecnica. Per esempio, il problema che coinvolge la relazione fra padre e figlio, può essere formalizzato con il termine

```
{padre(Nfiglio,Cognome,Npadre)}
```

e, le sue eventuali istanze possono evidentemente essere risolte utilizzando la tabella con i dati anagrafici già vista

```
{nascita(Nfiglio,Npadre,Cpadre,Nmadre,Cmadre)}.
```

Il legame fra il problema in esame e la tabella utilizzabile per la sua soluzione viene descritto con la seguente regola

```
padre(Nfiglio,Cognome,Npadre) :-
   nascita(Nfiglio,Npadre,Cognome,_,_).
```

In modo analogo si può procedere per descrivere la relazione fra madre e figlio.

```
madre(Nfiglio,Cognome,Nmadre,Cmadre) :-
   nascita(Nfiglio,_,Cognome,Nmadre,Cmadre).
```

Con la riscrittura si crea virtualmente una nuova tabella e si rende quindi possibile la soluzione di un problema mediante l'utilizzo di una tabella già descritta in modo esplicito: la tabella virtuale contiene solo le colonne strettamente necessarie per il problema in esame; i nomi delle entità che non servono vengono sostituiti dal segno "_".

La riscrittura, in generale si rende quindi opportuna quando, formalizzato il problema da risolvere, ci si rende conto che le informazioni necessarie per risolvere questo problema sono contenute in una tabella già precedentemente formalizzata; ovvero quando le colonne della tabella che sarebbe necessaria per questo nuovo problema sono contenute in una precedente tabella. Nella riscrittura, non è necessario usare i nomi delle variabili già utilizzati, si deve solo usare il nome della tabella già costruita e rispettare l'ordine degli argomenti (cioè delle colonne): in particolare, nel caso dello stato civile, la cui definizione implicita già definita è la seguente

```
{nascita(Nome,Npadre,Cpadre,Nmadre,Cmadre)},
```

i problemi di definire il padre e la madre di una persona, potevano essere risolti indifferentemente con questa scrittura

```
padre(Figlio,Cognome,Padre) :-
   nascita(Figlio,Padre,Cognome,_,_).
madre(Figlio,Cognome,Madre,Cmadre) :-
   nascita(Figlio,_,Cognome,Madre,Cmadre).
```

o anche con la seguente e tutte le similari

```
padre(X, Y, Z) :-
   nascita(X, Z, Y,_,_).
madre(A, B, C, D) :-
   nascita(A, _, B, C, D).
```

In particolare, in quest'ultima scrittura, `A`, `B`, `C` e `D` rappresentano rispettivamente il nome del figlio il cognome del padre (e quindi anche del figlio), il nome della madre e il cognome della madre perché nel termine usato per la descrizione esplicita della tabella `nascita` che riporta i dati di stato civile al primo, terzo, quarto e quinto posto si trovano rispettivamente queste informazioni.

Nonostante sia sintatticamente possibile usare nomi di variabili come `A`, `B`, `X` o `Y`, si ribadisce che questa è una pratica assolutamente da evitare: si ribadisce il consiglio di seguire la convenzione di usare stringhe che, in linguaggio naturale, evocano il significato delle rispettive entità.

Scomposizioni sequenziali in due sottoproblemi

I problemi non primitivi (cioè quelli che non possono essere risolti consultando una tabella o con una semplice formula aritmetica) vengono risolti mediante scomposizione in sequenze di sottoproblemi primitivi; nei casi più semplici si tratta di individuare due o più tabelle da consultare in sequenza o di usare più formule. I ogni caso, le informazioni di input non sono sufficienti per trovare direttamente l'output. La scomposizione deve mettere in evidenza (almeno) due sottoproblemi:

- il primo sottoproblema, deve essere risolubile con i soli dati iniziali e mette a disposizione informazioni aggiuntive;
- il secondo sottoproblema fornisce l'output richiesto e deve essere risolubile con (parte de) i dati iniziali e le informazioni aggiuntive trovate risolvendo il primo sottoproblema.

Per esempio, per comunicare via telefono (Telefono) ad un compagno di viaggio (Nome) l'ora della partenza di un treno (Ora) per una certa destinazione (Destinazione), si deve prima consultare l'orario ferroviario (e trovare l'ora di partenza per la destinazione in oggetto), poi consultare l'elenco telefonico (e trovare il numero da comporre per comunicare con il compagno di viaggio).

La formalizzazione dei tre problemi citati può essere la seguente

```
{comunicare(Nome, Destinazione, Ora, Telefono)}
{orario(Destinazione, Ora)}
{telefono(Nome,Tel)}
```

In questo caso si è usata la stringa `telefono` per indicare l'elenco telefonico e la stringa `Tel` per indicare il numero telefonico.

La scomposizione che lega logicamente i tre problemi è data dalla seguente regola

```
comunicare(Nome, Destinazione, Ora, Tel) :-
   orario(Destinazione, Ora), telefono(Nome,Tel).
```

Questa scomposizione si legge nel modo seguente:

il problema di trovare il numero di telefono (`Tel`) di una persona (`Nome`) cui comunicare l'orario di partenza (`Ora`) per una certa destinazione (`Destinazione`) è equivalente a risolvere i due (sotto)problemi seguenti

- trovare l'orario di partenza (`Ora`) per una data destinazione (`Destinazione`) e
- trovare il numero di telefono (`Tel`) della persona (`Nome`).

Quello appena illustrato è un esempio di scomposizione sequenziale di un problema in due sottoproblemi: il problema dato viene scomposto in una sequenza di due sottoproblemi che, risolti nell'ordine esplicitato dalla regola, conducono alla soluzione del problema di partenza.

Altro esempio di scomposizione di problema in due sottoproblemi: trovare i nomi dei genitori dell'amico che fa l'avvocato.

Il problema dato si può risolvere mediante i due sottoproblemi: trovare il nome della persona sapendo che fa l'avvocato e trovare i genitori di questa persona sapendone ora il nome.

Il problema dato può essere formalizzato come

```
{p(Nome-padre, Nome-madre, avvocato)}
```

e, ipotizzando di poter usare tabelle già viste per la nostra agenda

```
{agenda(Nome, Cognome, Telefono, Professione)}
```

e per lo stato civile

```
{nascita(Nome, Npadre, Cpadre, Nmadre, Cmadre)}
```

la scomposizione si può formalizzare nel modo seguente

```
p(Nome-padre, Nome-madre, avvocato) :-
   agenda(Nome, Cognome, _, avvocato),
   nascita(Nome, Nome-padre, Cognome, Nome-madre, _).
```

Si ricorda che, quando si riutilizza una formalizzazione già definita, per il nome del problema si deve usare la stessa stringa già usata, gli argomenti (o entità) devono essere gli stessi e vanno elencati nel medesimo ordine, mentre non è necessario, per questi ultimi, usare la medesima stringa identificativa (usata nella formalizzazione).

La scomposizione precedente può quindi anche essere scritta nel modo seguente

```
p(A,B,avvocato) :- agenda(X,Y,_,avvocato), nascita(X,A,Y,B,_).
```

Ribadendo la validità dell'osservazione che i nomi devono essere stringhe «significative», in questa formulazione, `X` è il nome dell'avvocato perché nella tabella agenda il primo argomento è il nome della persona che, nella medesima riga di quella tabella, ha il numero telefonico al terzo posto e la professione al quarto; così `A` e `B` sono i nomi dei genitori dell'avvocato perché nella tabella nascita dello stato civile il secondo (`A`) e il quarto argomento (`B`) sono i nomi dei genitori della persona il cui nome (`X`) è riportato nel primo argomento; il cognome comune di padre e figlio (cioè `Y`) è riportato in seconda posizione nell'agenda e in terza nello stato civile.

E' lasciata al lettore la soluzione del corrispondente problema generico.

In questo esempio viene illustrato un **problema di decisione**: in questi problemi la risposta è SÌ oppure NO, a seconda che esista o meno, tra le entità coinvolte nel problema, una determinata relazione.

Esempio: Dati i nomi di due persone, verificare se sono fratelli.

In questo problema la risposta è SÌ oppure NO, a seconda che esista o meno la relazione di fratellanza fra le due persone indicate.

La formalizzazione del problema può essere rappresentata dal seguente termine, tenuto conto che i due fratelli avranno il medesimo cognome e nomi diversi.

```
{fratelli(Nome-1, Nome-2, Cognome)}.
```

Poiché non è ragionevole pensare che si possa costruire una tabella contenente tutte le coppie di fratelli, il problema posto non è primitivo.

Una scomposizione ragionevole del problema è la seguente:

- trovare i nomi dei genitori della prima persona,
- verificare che i nomi dei genitori della seconda persona sono uguali a quelli della prima.

Questi due sottoproblemi sono tutti primitivi e si risolvono consultando la tabella `nascita`. Pertanto, la scomposizione del problema dato può avere la forma seguente

```
fratelli(Nome-1, Nome-2, Cpadre) :-
   nascita(Nome-1, Npadre, Cpadre, Nmadre, Cmadre),
   nascita(Nome-2, Npadre, Cpadre, Nmadre, Cmadre).
```

Il primo sottoproblema viene risolto trovando i nomi dei genitori della prima persona; il secondo verifica se nella tabella dello stato civile esiste una persona (`Nome-2`) che ha quei genitori.

Il backtracking

Viene ora illustrato un particolare comportamento del sistema, detto ***backtracking***, che consente di ottenere soluzioni multiple. Siano date le tabelle {professione(Nome,Cognome,Professione)} e {corsoDiLingua(Nome,Cognome,Lingua)} che descrivono la professione e la lingua straniera studiata per un dato insieme di persone. Il problema di trovare nome e cognome di una persona e la lingua studiata assegnata la professione può essere risolto con il seguente programma che contiene le due tabelle esplicite e la regola che descrive la scomposizione del problema in due sottoproblemi (risolvibili con interrogazione delle due tabelle).

```
professione(paola,verdi,ingegnere).
professione(aldo,rossi,avvocato).
professione(gianni,bianchi,avvocato).
professione(maria,bruni,notaio).
professione(antonio,neri,avvocato).
professione(mario,biondi,medico).

corsoDiLingua(aldo,rossi,inglese).
corsoDiLingua(anna,verdi,tedesco).
corsoDiLingua(paola,verdi,francese).
corsoDiLingua(antonio,neri,russo).
corsoDiLingua(antonio,neri,tedesco).
corsoDiLingua(mario,biondi,greco).
corsoDiLingua(mario,biondi,inglese).
corsoDiLingua(maria,bruni,cinese).

problema(Professione,Nome,Cognome,Lingua) :-
```

```
professione(Nome,Cognome,Professione),
corsoDiLingua(Nome,Cognome,Lingua).
```

Un esempio di interazione con il sistema è la seguente.

```
?- problema(avvocato,Nome,Cognome,Lingua) .
Nome=aldo, Cognome=rossi, Lingua=inglese ;
Nome=antonio, Cognome=neri, Lingua=russo ;
Nome=antonio, Cognome=neri, Lingua=tedesco ;
No
```

La prima risposta si ottiene risolvendo il primo sottoproblema consultando la tabella delle professioni e fermandosi alla seconda riga e risolvendo il secondo consultando la tabella della lingua fermandosi alla prima riga.

La seconda risposta, che si richiede digitando il punto e virgola, si ottiene cercando di risolvere il secondo sottoproblema continuando a consultare la tabella della lingua dalla seconda riga in poi; poiché non ci sono altre soluzioni (non ci sono altre lingue studiate da `Nome = aldo` e `Cognome = rossi`) il sistema cerca un'altra soluzione innescando il ***backtracking***, cioè cercando una ulteriore soluzione per il primo sottoproblema; ciò avviene continuando a consultare la prima tabella dalla terza riga in poi. Trovata una nuova soluzione per il primo sottoproblema (`Nome = giovanni` e `Cognome = bianchi`) il sistema cerca di risolvere il secondo sottoproblema riprendendo la consultazione della seconda tabella a partire dall'inizio; non esistendo alcuna soluzione del secondo sottoproblema (la persona `Nome = giovanni` e `Cognome = bianchi` non studia alcuna lingua), il sistema cerca (con il ***backtracking***) una ulteriore nuova soluzione per il primo sottoproblema. Trovata la soluzione `Nome = antonio`, `Cognome = neri`, ad essa corrisponde, nella seconda tabella la soluzione `Lingua = russo` (quarta riga).

La terza soluzione si ottiene dalla riga successiva (`Lingua = tedesco`).

Poiché il ***backtracking*** non riesce a trovare altre soluzioni (vedi animazione), il sistema termina con la risposta No. Per attivare l'animazione, si deve prima scegliere la velocità di esecuzione cliccando su uno dei tre bottoni che riportano le velocità (per la prima volta si

suggerisce di scegliere lenta); quindi si deve cliccare il bottone con il punto interrogativo. Per vedere le soluzioni successive si deve ciccare il bottone con il punto e virgola.

Programma	Dati
problema(Professione,Nome,Cognome,Lingua) :- professione(Nome,Cognome,Professione), corsoDiLingua(Nome,Cognome,Lingua).	**professione** professione(paola,verdi,ingegnere). professione(aldo,rossi,avvocato). professione(gianni,bianchi,avvocato). professione(maria,bruni,notaio). professione(antonio,neri,avvocato). professione(mario,biondi,medico).
==> Premi il bottone di interrogazione! <== problema(avvocato,Nome,Cognome,Lingua) . Nome=aldo, Cognome=rossi, Lingua=inglese Nome=antonio, Cognome=neri, Lingua=russo Nome=antonio, Cognome=neri, Lingua=tedesco no scegli la Velocità di esecuzione:	**corsoDiLingue** corsoDiLingua(aldo,rossi,inglese). corsoDiLingua(anna,verdi,tedesco). corsoDiLingua(paola,verdi,francese). corsoDiLingua(antonio,neri,russo). corsoDiLingua(antonio,neri,tedesco). corsoDiLingua(mario,biondi,greco). corsoDiLingua(mario,biondi,inglese). corsoDiLingua(maria,bruni,cinese).

Ricapitolando, il *backtracking* viene usato quando si deve risolvere una sequenza di sottoproblemi. I sottoproblemi vengono risolti nell'ordine; quando la soluzione di un sottoproblema fallisce si cerca una nuova soluzione al precedente; risolto questo, si riprendono i tentativi di risolvere il successivo.

Scomposizioni sequenziali in tre o più sottoproblemi

In generale, un problema non primitivo può essere scomposto in due o più sottoproblemi. Nel paragrafo precedente sono state illustrate scomposizioni in due sottoproblemi, ora vengono presentate scomposizioni in tre o più sottoproblemi.

Si consideri il seguente problema:

nell'ambito delle nostre amicizie, i cui dati siano elencati nella tabella agenda, trovare il nome del fratello dell'avvocato.

Il testo del problema cita il nome da trovare e la professione del fratello, quindi la formalizzazione può essere la seguente

```
{fratello-di(avvocato,Fratello)}.
```

In questo caso la costante `avvocato` è l'input e la variabile `Fratello` è l'output. Non essendo ipotizzabile l'esistenza di una tabella da consultare direttamente per risolvere questo problema, si deve ricorrere ad una scomposizione.

Si dovrà trovare nell'ordine: il nome della persona che fa l'avvocato (utilizzando la tabella `agenda`), i nomi dei genitori di questa persona (utilizzando la tabella `nascita` dello stato civile) e, infine, il nome di un secondo figlio di questi genitori (utilizzando ancora la tabella `nascita`). Questi tre sottoproblemi possono quindi essere così formalizzati come

```
{agenda(Nome,Cognome, _ , avvocato)},
{nascita(Nome,Npadre,Cognome,Nmadre,Cmadre)},
{nascita(Fratello,Npadre,Cognome,Nmadre,Cmadre)}.
```

La scomposizione, infine, viene descritta dalla seguente regola

```
fratello-di(avvocato, Fratello) :-
   agenda(Nome, Cognome, _, avvocato),
   nascita(Nome, Npadre, Cognome, Nmadre, Cmadre),
   nascita(Fratello, Npadre, Cognome, Nmadre, Cmadre),
   Nome =/= Fratello.
```

In questa formulazione si deve verificare che la variabile `Fratello` sia sostituita con una costante diversa da quella con cui è sostituita la variabile `Nome`: ciò viene ottenuto mediante il simbolo `=/=` che significa appunto che l'entità di sinistra deve essere diversa da quella che sta a destra di questo segno.

Esercizio.

Spiegare in dettaglio la necessità dell'ultimo sottoproblema della scomposizione (`Nome =/= Fratello`) appena illustrata; determinare cosa succede se questo sottoproblema (che rappresenta un vincolo che deve essere rispettato) viene omesso.

Esercizio.

Risolvere il problema precedente nella sua versione generica.

Esempio

Esempio di scomposizione di un problema in sottoproblemi di tipo aritmetico: calcolare l'area della superficie totale di un cilindro noti il raggio R della base e l'altezza H.

Per calcolare l'area della superficie totale di un cilindro si deve prima trovare l'area delle basi, poi trovare l'area della superficie laterale. Se questi tre problemi vengono formalizzati nel modo seguente

```
{superficie-tot(R,H,At)},
{area-basi(R,A-basi)},
{area-lat(R,H,A-lat)}.
```

La scomposizione viene espressa dalla seguente regola

```
superficie-tot(R,H,At) :-
   area-basi(R,A-basi),
   area-lat(R,H,A-lat),
   At = A-basi + A-lat.

area-basi(R,A-basi) :- A = 3.14×R**2, A-basi = 2×A.

area-lat(R,H,A-lat) :-
   Circonferenza = 2×3.14×R,
   A-lat = Circonferenza×H.
```

Per illustrare la flessibilità con cui può essere descritta la scomposizione di un problema in sotto problemi, vengono riportate altre due scomposizioni alternative del medesimo problema.

Prima alternativa.

```
superficie-tot(R,H,At) :-
   area-basi(R,A-basi),
```

```
    area-lat(R,H,A-lat),
    somma(A-basi, A-lat, Atot).

area-basi(R,A-basi) :- A = 3.14*R**2, A-basi = 2*A.

area-lat(R,H,A-lat) :-
    Circonferenza = 2*3.14*R,
    A-lat = Circonferenza*H.

somma(A-basi, A-lat, Atot) :- At = A-basi + A-lat.
```

Seconda alternativa.

```
superficie-tot(R,H,At) :-
    A = 3.14*R**2, A-basi = A + A,
    Circonferenza = 2*3.14*R,
    A-lat = Circonferenza*H,
    At = A-basi + A-lat.
```

Scomposizione successive

Percorsi stradali

Riprendiamo il problema di trovare un elenco che descrive il percorso stradale tra due città, già in precedenza formalizzato come segue,

```
{percorso(Città-di-partenza, Città-destinazione, Elenco)}.
```

La conoscenza necessaria per risolvere questo problema è rappresentata dalla tabella in cui sono descritti tutti i tronchi stradali utilizzabili per costruire i vari percorsi. La descrizione implicita di questa tabella può essere la seguente

```
{tr(Prima-città,Seconda-città)}.
```

La tabella esplicita (relativa ad una porzione dei collegamenti stradali di città italiane) può essere la seguente (le città sono individuate dalle rispettive sigle automobilistiche: in questo elenco fc sta per Cesena)

```
tr(mi,pc).

tr(mi,to).

tr(bs,pc).

tr(mi,ge).

tr(pr,sp).

tr(sp,lu).

tr(mi,bs).

tr(pc,al).

tr(pr,pc).

tr(lu,fi).

tr(fi,bo).

tr(pr,mo).

tr(bo,fe).

tr(bo,fc).

tr(rn,rsm).
```

```
tr(ge,sp).
tr(ge,sv).
tr(sv,to).
tr(to,al).
tr(fe,pd).
tr(pd,vr).
tr(vr,mo).
tr(mo,bo).
tr(vr,bs).
tr(pd,ve).
tr(vr,bz).
tr(rn,fc).
tr(rn,rsm).
tr(rn,fe).
tr(fe,vr).
tr(mi,al).
```

Con riferimento a questa tabella, si giustificano direttamente le seguenti interazioni: la prima risolve il problema di trovare le città cui si perviene percorrendo un tronco a partire da Milano; la seconda esegue la medesima ricerca a partire da Bologna.

```
?- tr(mi,X).
X = pc;
X = to;
X = ge;
X = bs;
NO
?- tr(bo,X).
X = fe;
X = fc;
```

```
NO
```

In questa seconda interazione, mancano i collegamenti di Bologna con Modena e con Firenze (nonostante i rispettivi tronchi siano presenti nella tabella); ciò avviene perché la domanda fa riferimento a collegamenti che partono da Bologna, mentre nella tabella quelli mancanti sono rappresentati da collegamenti che partono rispettivamente da Firenze e da Modena.

Se si vuole esplicitamente affermare che i collegamenti descritti in una tabella sono da intendersi in maniera bidirezionale (cioè a doppio senso di marcia), allora si deve esprimere il fatto che per ogni tronco da X a Y si deve supporre presente anche il tronco da Y a X. Questo risultato si ottiene mediante le regole seguenti (che, per riscrittura, costruiscono la tabella virtuale `{tr1(Prima-città,Seconda-città)}`

```
tr1(X,Y) :- tr(X,Y).

tr1(X,Y) :- tr(Y,X).
```

In questo modo, `tr(X,Y)` rappresenta un tronco a senso unico (dalla prima città verso la seconda), mentre `tr1(Y,X)` rappresenta un tratto a doppio senso di marcia; infatti, in nella tabella tr1 il tratto da `X` a `Y` esiste sia quando in `tr` è presente il medesimo tratto sia quando è presente il tratto da `Y` a `X` (quindi in tr1 è garantito il doppio senso di marcia). Ricordando quanto detto a proposito della riscrittura, si può affermare che `{tr1(Prima-città,Seconda-città)}` rappresenta una tabella virtuale dedotta da `{tr(Prima-città,Seconda-città)}e` nella quale tutti i tronchi sono a doppio senso di marcia; infatti `tr1(bo,mo)` risulta ora presente nella tabella virtuale perché in quella reale è presente il tratto `tr(mo,bo)`.

Ripetendo l'interrogazione precedente, ma utilizzando il termine `tr1` definito mediante riscrittura della tabella `tr`, si ottiene quindi la seguente interazione

```
?- tr1(bo,X).

X = fe;

X = fc;

X = fi;

X = mo;

NO
```

Utilizzando i collegamenti stradali descritti dal termine `tr1` vengono ora illustrati alcune soluzioni di problemi ottenute mediante scomposizioni successive.

Esempio 1.

Trovare un percorso tra due città X e Y composto da due tronchi.

Il problema può essere formalizzato come

```
{percorso2(X,Z,Y)}.
```

La scomposizione del problema è quindi rappresentata dalla seguente regola

```
percorso2(X,Z,Y) :- tr1(X,Z), tr1(Z,Y).
```

Per risolvere il problema di trovare un percorso tra Milano e Parma composto da due tronchi, si deve porre la seguente domanda ottenendo la relativa risposta

```
?- percorso2(mi,Z,pr).

Z = pc;

NO.
```

Esempio 2.

Trovare un percorso tra due città X e Y composto da 3 tronchi. In questo caso si desume quindi facilmente che la scomposizione del problema può avere la forma seguente

```
percorso3(X,Z,W,Y) :- tr1(X,Z), tr1(Z,W), tr1(W,Y).
```

Il lettore è invitato ad individuare, in questa scomposizione, una possibile «anomalia» (una città potrebbe comparire più volte); un suggerimento utile per individuarla è contenuto nell'esercizio seguente.

Esercizio

Trovare tutti i possibili percorsi tra Ferrara e Bologna con due tappe intermedie. (Oltre ai due percorsi legittimi `[fe,vr,mo,bo]` e `[fe,rn,fc,bo]`, esistono almeno altri 6 cammini «anomali» che passano più volte per una stessa città).

Esercizio

Il lettore descriva le scomposizioni di problemi relativi alla ricerca di percorsi composti rispettivamente da 4, 5 o 6 tronchi (rilevando eventuali anomalie).

Ragionamenti

Assegnati dei tratti stradali, gli esempi sopra trattati mostrano come possano essere costruiti dei percorsi. In modo del tutto analogo, assegnate semplici regole di deduzione del tipo se… allora… è possibile costruire una sequenza argomentativa. Supponendo di dover rappresentare un insieme di regole di deduzione è opportuno utilizzare il termine seguente con tre entità in modo da poter specificare rispettivamente il numero identificativo della regola, la sua premessa e la relativa conclusione

```
{regola(Numero,Premessa,Conclusione)}
```

Indicando i singoli enunciati con lettere dell'alfabeto, una tabella esplicita di regole di deduzione può avere la forma seguente

```
regola(1,a,b).

regola(2,a,c).

regola(3,a,d).

regola(4,b,c).

regola(5,b,e).

regola(6,b,f).

regola(7,c,g).

regola(8,c,h).

regola(9,d,i).

regola(10,d,j).

regola(11,e,k).

regola(12,e,m).

regola(13,f,n).
```

Ora, utilizzando questa tabella (che è definibile anche come base di conoscenza) è possibile risolvere problemi come i seguenti.

L'enunciato f è deducibile dall'enunciato g?

Ipotizzando che sia vero l'enunciato h, è vero l'enunciato n?

Con quale argomentazione è possibile giustificare l'enunciato m a partire dall'ipotesi b?

Questi problemi (del tutto simili a quello di trovare, se esiste, un percorso tra due città appartenenti a un dato grafo stradale) saranno risolti **in generale** nei capitoli seguenti; ora, in

analogia a quanto illustrato sopra per trovare collegamenti stradali di **lunghezza limitata e predefinita**, il lettore è invitato a risolvere gli esercizi seguenti.

Esercizi

1. Dato un enunciato (conseguente), trovare la regola e l'antecedente che lo giustificano.

2. Dati due enunciati, verificare se è possibile dedurre il secondo dal primo utilizzando due regole di deduzione.

3. Dati tre enunciati, trovare le regole che consentono di dedurre il secondo dal primo e il terzo dal secondo.

Calcoli aritmetici

Nei due paragrafi precedenti si è mostrato come il problema di trovare un percorso stradale sia simile a quello di trovare una argomentazione deduttiva. Supponendo di rappresentare formule numeriche del tipo y = f(x) con strutture simili a quelle usate per rappresentare i tratti stradali e le regole di deduzione, anche il problema di trovare un procedimento numerico consistente in una sequenza di formule (dove x è l'antecedente e y il conseguente) viene risolto con programmi dello stesso tipo.

Se la base di conoscenza è la seguente

```
formula(1,a,b).

formula(2,a,c).

formula(3,a,d).

formula(4,b,c).

formula(5,b,e).

formula(6,b,f).

formula(7,e,g).
```

il problema di trovare le tre formule per calcolare Z conoscendo X si risolve con la seguente regola

```
procedimento(X,Z,N1,N2,N3) :-
        formula(N1,X,Y), formula(N2,Y,W), formula(N3,W,Z).
```

Con questo programma (base di conoscenza e regola), si ottiene per esempio la seguente interazione

```
?- procedimento(a,g,N1,N2,N3).

N1 = 1, N2 = 5, N3 = 7;

NO
```

Una generalizzazione

Nelle applicazioni, le regole di deduzione del tipo se… allora…. possono avere più di un antecedente; in questo caso la loro formalizzazione ne deve tener conto. Questo argomento viene ora affrontato nel caso che le regole abbiano tutte due antecedenti (o premesse) e sempre un solo conseguente. La formalizzazione implicita di una tabelle di queste regole può essere la seguente

```
{regola(N, Ant1, Ant2, Conseguente}.
```

Un esempio di tabella esplicita (nella quale gli enunciati sono simbolicamente rappresentati da lettere dell'alfabeto) è la seguente

```
regola(1,a,b,c).

regola(2,a,c,e).

regola(3,a,d,h).

regola(4,b,c,g).

regola(5,b,e,g).

regola(6,b,f,h).

regola(7,c,g,i).

regola(8,c,h,j).

regola(9,d,i,g).
```

Per esempio, la regola 4 formalizza la seguente deduzione:

se (sono veri gli enunciati) b e c allora si può dedurre g.

Con questa base di conoscenza sono risolubili problemi del tipo seguente:

- Esiste una regola per dedurre l'enunciato h?
- Quali sono le regole e le relative premesse per dedurre g?

La domanda da porre al sistema per risolvere il primo problema (e le relative risposte del sistema) è la seguente:

```
?- regola(N,_,_,h).

N = 3 ;

N = 6 ;

NO
```

Nella tabella sono infatti presenti le regole 3 e 6 che consentono di dedurre h a partire rispettivamente da a,d e da b,f. La domanda da porre al sistema per risolvere il secondo problema (e le relative risposte del sistema) è la seguente:

```
?- regola(N, Ant1, Ant2, g).

N = 4, Ant1 = b, Ant2 = c ;

N = 5, Ant1 = b, Ant2 = e ;

N = 9, Ant1 = d, Ant2 = i ;

NO
```

Questa generalizzazione è facilmente estensibile alle formule numeriche che prevedono il calcolo di una quantità dipendente dai valori di due entità: per esempio il calcolo dell'area di un rettangolo conoscendo le misure della base e dell'altezza. In casi particolari, le regole e le formule possono essere contraddistinte da nomi invece che da numeri. Ricordando le nozioni di geometria elementare relative al triangolo rettangolo, è facile dare significato alla porzione di tabella esplicita sotto riportata, corrispondente alla seguente formalizzazione implicita `{formula(Nome, Risultato, Dato1, Dato2)}`.

```
formula(pitagora, ipotenusa, cateto1, cateto2).

formula(pitagora, cateto1, ipotenusa, cateto2).

formula(pitagora, cateto2, ipotenusa, cateto1).

formula(euclide, altezza, proiezione1, proiezione2).

formula(euclide, proiezione1, altezza, proiezione2).

formula(euclide, proiezione2, altezza, proiezione1).

formula(area1, area, altezza, ipotenusa).
```

```
formula(area1, altezza, area, ipotenusa).
formula(area1, ipotenusa, altezza, area).
```

La regola riportata in quinta riga formalizza la seguente conoscenza

Utilizzando il teorema di Euclide, è possibile calcolare la misura della proiezione del primo cateto sull'ipotenusa se sono assegnate le misure dell'altezza e della proiezione del secondo cateto.

L'interazione completa corrispondente al problema di conoscere come sia possibile calcolare l'altezza di un triangolo rettangolo è la seguente

```
?- formula(N, altezza,D1,D2).
N = euclide, D1 = proiezione1, D2 = proiezione2 ;
N = area1, D1 = area, D2 = ipotenusa ;
NO
```

La prima risposta dice che è possibile calcolare la misura dell'altezza (di un triangolo rettangolo) mediante la formula di Euclide se si conoscono le misure delle due proiezioni dei cateti sull'ipotenusa; la seconda individua la formula per calcolare l'area se si conosce l'area del triangolo e la misura dell'ipotenusa

La risposta NO (interpretabile come "non esistono altri modi per calcolare l'altezza") è relativa alla tabella esplicita sopra riportata. Aggiungendo alla tabella altre formule note dalla geometria, la risposta del sistema diventerebbe più articolata.

SCOMPOSIZIONI CONDIZIONALI

I predicati di controllo

I problemi finora illustrati, primitivi e non primitivi, hanno una caratteristica: ammettono un procedimento risolutivo, esplicitato dalla applicazione di un unico insieme di formule. Esistono problemi (primitivi e non primitivi) che ammettono procedimenti di soluzione diversi da scegliere volta per volta in funzione di particolari circostanze. Queste situazioni vengono descritte con tanti procedimenti quante sono le diverse situazioni possibili e, a ciascun procedimento, viene associato un predicato che deve risultare vero quando deve essere applicato quel procedimento. Queste scomposizioni si dicono condizionali.

Un predicato è una affermazione che contiene alcune variabili: questa affermazione (cioè il predicato) risulta vera per alcuni valori delle variabili e falsa per altri.

Prendendo a riferimento un problema generico (con due entità in input e una in output) formalizzato con la struttura `{problema(X,Y,Z)}`, la scomposizione condizionale (per esempio con quattro alternative) si scrive

```
problema(X,Y,Z) :- predicato1(X,Y), procedimento1(X,Y,Z).
problema(X,Y,Z) :- predicato2(X,Y), procedimento2(X,Y,Z).
problema(X,Y,Z) :- predicato3(X,Y), procedimento3(X,Y,Z).
problema(X,Y,Z) :- predicato4(X,Y), procedimento4(X,Y,Z).
```

Il predicato di controllo può quindi essere interpretato come una guardia che consente di applicare un procedimento solo se il predicato è vero. Quando le guardie sono mutuamente escludentesi, per ogni istanza del problema da risolvere viene di fatto reso applicabile uno solo dei procedimenti descritti dal programma. I predicati usati in queste scomposizioni devono essere decidibili, cioè per ogni assegnazione di valori per i parametri i loro valori di verità sono determinabili in un tempo finito.

Scomposizioni esplicite

Sia dato il seguente problema: calcolare i valori di una quantità Z definita nel modo seguente

$Z = X^2 + Y$ quando è vero il predicato $X < Y$,

$Z = (X + Y)^2$ quando è vero il predicato $X = Y$ e

$Z = X + Y^2$ quando è vero il predicato $X > Y$.

La formalizzazione del problema può essere la seguente `{problema(X,Y,Z)}`; casi di prova significativi, in questo caso, devono consentire di verificare tutte le alternative: un buon esempio può essere il seguente

```
{?- problema(1,2,3)}.

{?- problema(3,1,4)}.

{?- problema(1,1,4)}.
```

La scomposizione di questo problema deve evidentemente prevedere le tre alternative diverse, ciascuna delle quali deve essere applicata quando risulta vero il rispettivo predicato; il programma, ricordando lo schema generale visto nel paragrafo precedente, è quindi dato dalle seguenti tre regole

```
problema(X,Y,Z) :- X < Y, Z = X**2 + Y.

problema(X,Y,Z) :- X = Y, Z = (X + Y)**2.

problema(X,Y,Z) :- X > Y, Z = X + Y**2.
```

Poiché i tre predicati sono mutuamente escludentisi (cioè per ogni coppia di valori per X e Y solo uno dei tre predicati risulta vero), non ha alcuna importanza l'ordine in cui le tre regole vengono scritte. Posta la domanda

```
?- problema(2,1,W).
```

Il sistema cerca di risolvere questa istanza con la prima regola: il tentativo fallisce perché in questo caso non è vero il predicato $X < Y$; analogamente fallisce il tentativo con la seconda regola perché non è vero il predicato $X = Y$; il tentativo ha successo con la terza regola perché in questo caso il predicato $X > Y$ è vero. Il problema viene quindi risolto applicando la formula $Z = X + Y^2$; l'interazione completa è quindi la seguente.

```
?- problema(2,1,W).

W = 3.
```

Un secondo esempio di scomposizione condizionale si incontra risolvendo il seguente problema: dati due numeri X e Y assegnare a Z il valore del più grande dei due dati. Adottando per questo problema la formalizzazione seguente `{massimo(X,Y,Z)}`, si hanno i seguenti casi di prova

```
{?- massimo(2,1,2)}.

{?- massimo(3,3,3)}.
```

```
{?- massimo(4,5,5)}.
```

Ciò premesso, si deduce immediatamente che il problema del massimo fra due numeri ammette le due alternative seguenti

Z = X quando è vero il predicato X >= Y e

Z = Y quando è vero il predicato X < Y

e la scomposizione si scrive quindi come

```
massimo(X,Y,Z) :- X >= Y, Z = X.

massimo(X,Y,Z) :- X < Y, Z = Y.
```

Questo programma può anche essere scritto in modo più conciso nella maniera seguente

```
massimo(X,Y,X) :- X >= Y.

massimo(X,Y,Y) :- X < Y.
```

Questa scrittura dice che quando X>=Y il terzo argomento viene posto uguale al primo e, quando X<Y, il terzo argomento viene posto uguale al secondo.

Scomposizioni implicite

Un ulteriore esempio di scomposizione condizionale è rappresentato dal seguente problema:

Trovare un percorso tra due città X e Y composto al più da tre tronchi stradali.

Questo problema ammette tre procedimenti di soluzione rappresentati rispettivamente dal numero di tronchi stradali utilizzabili per costruire il percorso cercato; pertanto, formalizzato il problema con la struttura `{percorso1-2-3(X,Y,Elenco)}`, si possono individuare i seguenti casi di prova (con riferimento ai tronchi stradali descritti dalla tabella `tr` e dal programma `tr1`).

```
{?- percorso1-2-3(bo,fe,[bo,fe])}

{?- percorso1-2-3(bo,rn,[bo,fc,rn])}

{?- percorso1-2-3(bo,rsm,[bo,fc,rn,rsm])}
```

La scomposizione, ricordando la soluzione di problemi illustrati nel paragrafo precedente, è la seguente

```
percorso1-2-3(X,Y,[X,Y]) :- tr1(X,Y).

percorso1-2-3(X, Y, [X,Z,Y]) :- percorso2(X,Z,Y).

percorso1-2-3(X, Y, [X,Z,W,Y]) :- percorso3(X,Z,W,Y).


percorso2(X,Z,Y) :- tr1(X,Z), tr1(Z,Y).

percorso3(X,Z,W,Y) :- tr1(X,Z), tr1(Z,W), tr1(W,Y).
```

In questo problema non sono presenti esplicitamente i predicati per individuare l'alternativa da usare perché questi sono impliciti nel primo membro: la prima alternativa, infatti, viene utilizzata (e può essere usata solo) quando le due città sono direttamente collegate da un solo tronco (l'elenco delle città attraversate contiene due città); la seconda solo quando è necessario transitare per una (e una sola) città intermedia (l'elenco contiene tre città) e la terza quando l'elenco è composto esattamente da tre tronchi (il percorso contiene quattro città).

Esercizi

Trovare le risposte alle seguenti domande, utilizzando i termini `tr` e `tr1` precedentemente introdotti.

```
?- percorso1-2-3(bo,fi,L).

?- percorso1-2-3(bo,pd,L).

?- percorso1-2-3(mi, rsm, L).
```

Verificare con specifiche istanze del problema

```
{percorso1-2-3(X,Y,Elenco)}
```

che è possibile che una istanza possa avere più soluzioni. Mostrare quindi che è possibile che due città sono raggiungibili con più percorsi diversi, tutti composti da un numero di tronchi compreso fra 1 e 3.

Deduzioni e procedimenti di calcolo

In modo analogo a quanto visto per la determinazione di percorsi, è possibile mostrare che possono esserci linee argomentative diverse per giustificare un enunciato o procedimenti di calcolo differenti per risolvere un medesimo problema numerico. Prendendo in esame la seguente tabella contenete regole di deduzione, avente la medesima struttura formale di quella analoga sopra vista,

```
regola(1,a,b).

regola(2,a,c).

regola(3,a,d).

regola(4,b,c).

regola(5,b,e).

regola(6,b,f).

regola(7,c,g).

regola(8,c,h).

regola(9,d,i).

regola(10,d,j).

regola(11,f,g).

regola(12,c,d).

regola(13,d,h).
```

il programma per trovare una argomentazione composta al più da tre regole di deduzione per giustificare un enunciato Z a partire da una ipotesi X è il seguente

```
argomentazione3(X,Z,[N]) :- regola(N,Z,X).

argomentazione3(X,Z,[N1,N2])          :-          regola(N1,Y,X),
regola(N2,Z,Y).

argomentazione3(X,Z,[N1,N2,N3]) :-

      regola(N1,Y,X), regola(N2,W,Y), regola(N3,Z,W).
```

In modo analogo può essere ottenuto un programma per risolvere problemi mediante procedimenti che prevedano non più di tre formule e che utilizzino una base di conoscenza di formule del tipo seguente.

```
formula(1,a,b).
formula(2,a,c).
formula(3,a,d).
formula(4,b,c).
formula(5,b,e).
formula(6,b,f).
formula(7,e,g).
formula(8,g,f).
formula(9,f,c).
```

Il programma in questione (per trovare la lista delle regole da applicare per calcolare il valore di X conoscendo il valore Z, con non più di tre formule) è il seguente

```
procedimento3(X,Z,[N]) :- formula(N,Z,X).
procedimento3(X,Z,[N1,N2]) :-
      formula(N1,Y,X), formula(N2,Z,Y).
procedimento3(X,Z,[N1,N2,N3]) :-
      formula(N1,Y,X), formula(N2,W,Y), formula(N3,Z,W).
```

Utilizzando questi ultimi due programmi si ottengono, per esempio, le seguenti interazioni

```
?- argomentazione3(d,a,L).
L = [3];
L = [12,2];
L = [12,4,1];
NO

?- procedimento3(f,a,L).
L = [6,1];
NO

?- procedimento3(c,a,L).
```

```
L = [2];

L = [4,1];

L = [9,6,1];

NO
```

```
formula(pitagora, ipotenusa, cateto1, cateto2).

formula(pitagora, cateto1, ipotenusa, cateto2).

formula(pitagora, cateto2, ipotenusa, cateto1).

formula(euclide, altezza, proiezione1, proiezione2).

formula(euclide, proiezione1, altezza, proiezione2).

formula(euclide, proiezione2, altezza, proiezione1).

formula(area1, area, altezza, ipotenusa).

formula(area1, altezza, area, ipotenusa).

formula(area1, ipotenusa, altezza, area).
```

SCOMPOSIZIONI RICORSIVE

Le liste

La soluzione di molti problemi dipende dagli elementi contenuti in un insieme o riportati in un elenco; esempi tipici di questi problemi sono i seguenti:

- trovare il più grande fra gli elementi di un insieme (di numeri naturali);
- calcolare la somma degli elementi di un insieme;
- verificare se un nome è contenuto in un elenco.

Per scomporre problemi di questo tipo, è necessario disporre di una struttura che consenta di rappresentare in modo conciso (tutti gli elementi contenuti in) elenchi e insiemi. La struttura generale qui usata a questo fine è la **lista**. Una lista, già incontrata in precedenti paragrafi, può essere rappresentata in modo esplicito o implicito; il modo esplicito utilizzato in questo testo consiste nell'elencare tutti gli elementi contenuti nella lista racchiusi fra parentesi quadra e separati da virgole; i tre esempi che seguono illustrano il metodo esplicito:

```
[a,e,u,i,o];
```

[lunedì,martedì,mercoledì,giovedì,venerdì,sabato,domenica];

```
[1,2,3,5,7].
```

Il caso particolare di una lista che non contenga elementi (o di un insieme vuoto), si rappresenta come `[ ]`.

Il modo implicito per rappresentare una lista (non vuota) consiste nello specificare il primo elemento della lista e la lista di tutti gli elementi rimanenti: questo si ottiene con la scrittura

```
[T|C].
```

L'uguaglianza

```
[T|C] = [a,e,u,i,o]
```

impone che sia

```
T = a,
C = [e,u,i,o].
```

Gli elementi a sinistra e a destra del carattere speciale | (in questo caso rappresentati dalle variabili T e C) vengono detti rispettivamente *testa* e *coda* della lista. Si richiama l'attenzione sul fatto che una lista può anche essere rappresentata direttamente da una variabile: si è appena visto che la variabile C rappresenta appunto la lista di tutti gli elementi della lista delle vocali, meno a (la testa di quella lista).

Se [T|C] = [a], allora si ha T = a e C = [ ]; una lista vuota non contiene alcun elemento, quindi non ha senso chiedere quali siano la testa e la coda di una lista vuota!

La rappresentazione implicita di una lista viene usata, in generale, nella formalizzazione dei problemi e nella descrizione delle scomposizioni; la rappresentazione esplicita viene in genere usata nelle domande, cioè quando viene sottoposta la richiesta di soluzione di una istanza specifica di un problema.

L’appartenenza di un elemento a una lista

Una prima applicazione della struttura lista viene illustrata per risolvere il seguente problema:

verificare se un dato elemento è contenuto in una lista.

Il problema contiene due entità: un elemento e una lista; pertanto la formalizzazione del problema può essere rappresentata da

```
{appartenenza(Elemento, Lista)}.
```

Casi di prova significativi sono

```
{?- appartenenza(e,[a,e,u,i,o])} con risposta YES
{?- appartenenza(6,[1,2,3,5,7])} con risposta NO
{?- appartenenza(p,[ ])} con risposta NO
```

La soluzione del problema richiede un confronto fra l’elemento dato e (potenzialmente tutti) gli elementi della lista; quindi, non solo il problema non è primitivo, ma non può essere risolto né con una scomposizione sequenziale né con una condizionale in quanto queste scomposizioni comportano sempre un numero predeterminato di sottoproblemi, mentre in questo caso il numero di verifiche da fare cambia da caso a caso, a seconda della lunghezza della lista. La scomposizione deve quindi prevedere la possibilità che un certo sottoproblema (il confronto fra due elementi) debba essere risolto un numero (a priori) indeterminato di volte; sarà la specifica istanza a determinarne il numero effettivo.

Per ottenere la scomposizione di questo problema, si può fare riferimento alla seguente definizione di appartenenza:

- un elemento E appartiene a una lista L = [T|C] se è uguale alla testa T della lista oppure
- l'elemento E appartiene alla lista L = [T|C] se appartiene alla coda C di quella lista.

Da questa definizione si desume immediatamente la seguente scomposizione

```
appartenenza(E, [T|C]) :- E = T.
appartenenza(E, [T|C]) :- appartenenza(E,C).
```

Per mostrare che la descrizione della scomposizione sopra scritta è operativa, viene ora illustrato il processo svolto dal sistema per giungere alla risposta della seguente domanda

```
?- appartenenza(u,[a,e,u,i,o]).
```

Per trovare la risposta a questa domanda, il sistema può usare (solo) il seguente programma.

```
/* 1 */ appartenenza(E, [T|C]) :- E = T.
/* 2 */ appartenenza(E, [T|C]) :- appartenenza(E,C).
```

Se la domanda si unifica con la 1), la risposta è SÌ; se si unifica con la 2), la risposta viene rimandata alla soluzione del problema descritto nel corpo della regola. Inizialmente, l'unificazione con la 1) fallisce (infatti E = u e T = a, quindi non è vero che E = T. L'unificazione con la 2) comporta

```
appartenenza(u, [a|[e,u,i,o]]]) :- appartenenza(u,[e,u,i,o]).
```

Quindi il sistema sostituisce il problema originale con il seguente

```
?- appartenenza(u,[e,u,i,o]).
```

Di nuovo fallisce l'unificazione con la 1) e ha successo quella con la 2) producendo il seguente risultato

```
appartenenza(u, [e|[u,i,o]]]) :- appartenenza(u,[u,i,o]).
```

Il problema che deve ora essere risolto è il seguente

```
? - appartenenza(u,[u,i,o]).
```

Ora è possibile l'unificazione con la 1); infatti

```
appartenenza(u, [u|[i,o]]) :- u = u.
```

A questo punto, la verifica ha avuto esito positivo e quindi il sistema risponde SÌ.

La scomposizione appena vista per il problema dell'appartenenza di un elemento a una lista si dice ricorsiva perché in una delle regole usate (la seconda) la formalizzazione del problema viene usata sia come testa della regola sia inserita nel corpo; tuttavia, le due formalizzazioni si riferiscono a istanze diverse del problema: nella testa si fa riferimento alla lista originaria, mentre nel corpo viene utilizzata la coda di quella lista. Altro elemento caratterizzante le scomposizioni ricorsive è la presenza di una regola che esplicita la soluzione del problema nel caso di dimensione minima della lista. Queste due regole, sempre presenti nelle scomposizioni ricorsive si dicono rispettivamente **passo induttivo** e **passo base** della scomposizione.

Esercizi

Ricostruire il processo svolto dal sistema per rispondere alle domande seguenti e formulare le rispettive risposte.

```
?- appartenenza(k,[ ]).
```

```
?- appartenenza(x,[z,x,c]).
```

```
?- appartenenza(4,[3,5,7]).
```

La somma degli elementi di una lista

La somma degli elementi di una lista (1)

La scomposizione ricorsiva viene ora applicata per risolvere il problema di calcolare la somma S degli elementi contenuti nella lista [a_n, a_{n-1},…,a_2 ,a_1] contenente n numeri. Il problema coinvolge due entità: una lista L di numeri e la loro somma S; la formalizzazione del problema può dunque essere la seguente:

```
{somma(L,S)}.
```

Casi di prova significativi sono

```
{?- somma([2,4,5],11).}
```

```
{?- somma([ ],0).}
```

```
{?- somma([4],4).}
```

Anche in questo caso è facile constatare che il problema non è primitivo e non può essere scomposto né in modo sequenziale né in modo condizionale (il numero di addizioni da eseguire

dipende dalla dimensione della lista associata alla istanza specifica). Per individuare la scomposizione si può fare riferimento allo schema seguente

$S_0 = 0$

$S_1 = S_0 + a_1$

$S_2 = S_1 + a_2$

$S_3 = S_2 + a_3$

...

$S_n = S_{n-1} + a_n$

Da questo schema risultano immediate due conseguenze:

- se la lista non contiene elementi (n=0), la somma (S_0) deve essere zero (cioè, se L=[] allora S=0;
- se la lista contiene elementi (cioè ha dimensione n>0), la somma cercata (S_n) è uguale alla somma dei suoi primi n-1 elementi (S_n-1) più l'ultimo (a_n).

Ricordando la formalizzazione del problema e posto T = a_n e C = [a_{n-1},…,a_2, a_1], la scomposizione deducibile dal ragionamento sopra riportato è la seguente

```
somma([ ],0).
somma([T|C], Sn) :- somma(C, Sn-1), Sn = Sn-1+ T.
```

Esercizi

Utilizzando il ragionamento sopra esplicitato per una istanza del problema dell'appartenenza, ricostruire il processo svolto dal sistema per rispondere alle domande associate ai tre casi di prova.

La somma degli elementi di una lista (2)

La soluzione di un problema che richiede lo svolgimento di un ciclo (cioè la ripetizione di una medesima operazione per un appropriato numero di volte) può anche essere descritta da una scomposizione diversa da quella ricorsiva sopra illustrata. Una semplice applicazione di questo tipo di scomposizione, detta iterativa, viene ora illustrata per la a soluzione del problema appena visto di calcolare la somma di n numeri dati. Con questo procedimento, alternativo a quello ricorsivo sopra visto, il problema viene risolto con un programma che esplicita la esecuzione di tante addizioni (cioè la soluzione di tanti sottoproblemi) quanti sono i numeri da sommare. La scomposizione deve rendere esplicite due azioni:

- la prima per eseguire una addizione e procedere alla successiva e

- la seconda da eseguire quando il processo deve aver termine.

Anche queste scomposizioni iterative (come le ricorsive) prevedono quindi sempre (almeno) due alternative (l'aggiornamento e il criterio d'arresto) e per questo sono anche condizionali. Inoltre, poiché il risultato si ottiene per passi successivi (in questo caso le singole addizioni che aggiornano il valore della somma), si deve inizialmente stabilire anche il valore iniziale con cui far partire il processo (cioè per fare il primo aggiornamento); nel caso specifico, in particolare, il valore iniziale della somma (prima di sommare il primo addendo) deve essere 0. La formalizzazione del problema (che prevede tre entità: la lista contenente i numeri da sommare, la somma dei numeri, il valore iniziale della somma per fare il primo aggiornamento) è la seguente `{somma1(Lista,Somma,0)};` i casi di prova possono essere

```
{?- somma1([2,4,5],11,0)}

{?- somma1([ ],0,0)}

{?- somma1([4],4,0)}
```

La formalizzazione di questo problema può anche contenere solo le prime due entità, lasciando a una regola di riscrittura il compito di definire il valore iniziale per l'aggiornamento. Utilizzando questo suggerimento, la formalizzazione del problema e i rispettivi casi di prova possono essere riformulati come segue

```
{somma1(Lista,Somma)}

{?- somma1([2,4,5],11,0)}

{?- somma1([ ],0,0)}

{?- somma1([4],4,0)}
```

La scomposizione complessiva del problema si scrive con le seguenti tre clausole

```
/1/ somma1(Lista,Somma) :- somma2(Lista,Somma, 0).

/2/ somma2([Primo-elemento|Rimanenti],S ,P ) :-
       Q = P+Primo-elemento,
       somma2(Rimanenti, S,Q).

/3/ somma2([ ], S,S).
```

La prima clausola riscrive il problema aggiungendo una entità che rappresenta il valore con cui iniziare l'aggiornamento.

La seconda clausola esegue un singolo aggiornamento usando il primo elemento della lista e fa ripetere il procedimento sugli elementi rimanenti. (La seconda volta, il primo elemento sarà in realtà il primo dei rimanenti, e quindi il secondo della lista iniziale e così via di aggiornamento in aggiornamento fino a prendere in considerazione l'ultimo elemento; eseguito l'aggiornamento con l'ultimo elemento della lista, i rimanenti in realtà saranno una lista vuota che è rappresentata da [ ]).

La terza clausola descrive il passo finale e viene eseguita quando sono stati presi in esame tutti gli elementi e la lista originaria è stata vuotata; questo passo consiste nell'attribuire alla variabile risultato (il secondo argomento) il medesimo valore assunto dalla variabile che rappresenta l'aggiornamento (il terzo argomento).

Nella tabella seguente sono riportati via via i valori assunti dalle variabili nei passi che vengono eseguiti per calcolare la somma dei numeri contenuti nella seguente lista L=[7,9,2,5,1].

Passo	Clausola	Primo-elem	Rimanenti	S	Q
0	1: riscrittura	7	[9,2,5,1]	-	0
1	2: aggiornamento	7	[9,2,5,1]	-	7
2	2: aggiornamento	9	[2,5,1]	-	16
3	2: aggiornamento	2	[5,1]	-	18
4	2: aggiornamento	5	[1]	-	23
5	2: aggiornamento	1	[ ]	-	24
6	3: arresto	-	[ ]	24	24

La tabella contiene i valori assunti dalle variabili per calcolare la somma dei numeri contenuti nella lista [7,9,2,5,1]

La variabile S acquista un valore (cioè viene istanziata) solo quando viene eseguita la terza clausola; in questo caso Primo-elem non esiste perché il primo argomento è una lista vuota (quindi priva di primo elemento).

La struttura formale

La modalità delle scomposizioni illustrata in modo informale, mediante esempi, nelle pagine precedenti di questo capitolo, è detta ricorsività o ricorsione. Questa modalità si basa sull'utilizzo di una scomposizione multipla con almeno due alternative: almeno una delle regole fa riferimento allo stesso problema da risolvere (ovvero contiene nel corpo il problema stesso) e viene detta **passo induttivo** della ricorsione. La regola in cui non compare il problema da risolvere è detta **passo base** della ricorsione.

Nell'esempio precedente si ha una (sola) regola per il passo induttivo (in cui compare lo stesso problema che si sta definendo):

```
somma2([Primo-elemento|Rimanenti],S ,P ) :-
    Q = P+Primo-elemento,
    somma2(Rimanenti, S,Q).
```

e una (sola) regola per il passo base della ricorsione:

```
somma2([ ], S,S).
```

Ovviamente in una soluzione ricorsiva c'è sempre almeno una alternativa della scomposizione multipla che rappresenta il passo iduttivo (altrimenti non sarebbe una ricorsione, ma una semplice scomposizione multipla) è meno evidente che in una ricorsione ci deve *sempre* essere almeno una regola che rappresenta il passo base. Il passo ricorsivo, di fatto, trasforma la soluzione di una istanza di «ordine» *n* del problema dato nella soluzione di una istanza di «ordine» *n*-1 più (eventualmente) un altro sottoproblema (in questo caso il calcolo di una espressione aritmetica). La presenza del passo base è dovuta alla necessità di interrompere questa riduzione all'indietro della dimensione: il passo base infatti individua la dimensione minima (in genere zero o uno) e esplicita la soluzione del problema in quel caso.

Le due alternative della scomposizione multipla (passo base e passo induttivo) hanno come guardia o predicato di controllo implicito la dimensione della istanza specifica del problema che si sta risolvendo (in questo caso rappresentata dal numero di elementi della lista): una delle due alternative (il passo induttivo) può essere usata solo quando la lista non è vuota (deve esserci almeno un primo elemento); l'altra (il passo base) viene usata solo quando la lista è vuota.

Grafi e alberi

È esperienza comune la consultazione di una carta geografica che descriva i collegamenti stradali tra città. In questo contesto sono significativi gli archi (i collegamenti stradali) e i nodi (le città) da essi congiunti. Una struttura di questo tipo (insiemi di nodi e di archi che li congiungono) viene detta grafo. Se gli archi stradali sono tutti a doppio senso di marcia, il grafo è detto non orientato; se alcuni archi (o tutti) sono percorribili solo a senso unico, il grafo si dice orientato. Un grafo può essere rappresentato come una tabella nella quale si mette nella prima colonna il nodo di origine dell'arco (la città da cui parte il tronco stradale a senso unico) e nella seconda colonna il nodo d'arrivo dell'arco (la città in cui arriva il tronco stradale).

Si consideri ora una tabella che descrive percorsi stradali (a senso unico!) tra due città la cui descrizione implicita sia

```
{arco(Città1,Città2)}
```

e la cui descrizione esplicita sia

```
arco(f,g).
arco(a,f).
arco(a,b).
arco(b,c).
arco(b,d).
arco(d,i).
arco(d,h).
arco(a,e).
```

È facile verificare che questa tabella rappresenta un grafo orientato: in esso tutti gli archi sono a senso unico. Anche la tabella `{tr(Prima-città,Seconda-città)}` (già vista in precedenza nel paragrafo 4.3) rappresenta un grafo orientato; mentre la tabella `{tr1(Prima-città,Seconda-città)}`, ottenuta in modo virtuale dalla precedente mediante il programma

```
tr1(X,Y) :- tr(X,Y).
tr1(X,Y) :- tr(Y,X).
```

rappresenta un grafo non orientato (infatti, per ogni arco da `X` a `Y` presente nel grafo `tr` esistono due archi nel grafo `tr1`: quello da `X` a `Y` e quello da `Y` a `X`.

Esercizio

Verificare che il programma

```
tr2(X,Y) :- tr(X,Y).
tr2(Y,X) :- tr(X,Y).
```

Costruisce la medesima tabella virtuale (e quindi il medesimo grafo non orientato) costruita con il programma basato sul termine `tr1`.

Un percorso in un grafo può essere rappresentato da una lista contenente nell'ordine i nodi collegati a due a due da una arco. Nei grafi (orientati o non orientati) possono esistere cicli: un ciclo è rappresentato da un percorso che partendo da un nodo vi fa ritorno dopo essere passato per altri nodi. Il percorso `[bo,fe,pd,vr,mo,bo]` è un ciclo, sia nel grafo `tr`, sia in quello `tr1`; mentre

il percorso `[mi,pc,al,mi]` è un ciclo nel grafo `tr1`, mentre non lo è nel grafo `tr` (in quest'ultimo grafo infatti, gli archi sono orientati, cioè a senso unico e non consentono di raggiungere il nodo `mi` né dal nodo `al` né da quello `pc`).

Gli archi orientati hanno un nodo origine (il nodo da cui parte l'arco) e un nodo destinazione. Un grafo orientato nel quale ogni nodo destinazione è raggiungibile da uno e un solo nodo origine si dice albero. Il grafo sopra riportato e descritto dal termine {arco(Città1,Città2)} è un albero. (Vedi figura). In un albero deve esistere un nodo che non è destinazione di alcun arco: questo nodo si dice radice dell'albero (vedi nodo a della figura); i nodi dell'albero che non sono origine di alcun arco si dicono foglie dell'albero (vedi nodi e, c, h, g, i della figura). Il grafo rappresentato dalla seguente tabella

```
arco(f,g).
arco(a,f).
arco(a,b).
arco(b,c).
arco(b,d).
arco(d,i).
arco(d,h).
arco(a,e).
arco(h,i)
arco(g,c).
```

non è un albero (i nodi i e c sono raggiungibili per più vie; vedi figura).

Grafo con tutti i nodi dell’esempio

Nel disegno è riportata in modo grafico la rappresentazione dell’informazione che nel testo è descritta dalla tabella {arco(Città1,Città2)}. La direzione degli archi (che nel testo va dal primo al secondo argomento), nel grafico è indicata dalla direzione delle frecce, orientato dal nodo di partenza (il primo) al nodo di arrivo (il secondo).

Esercizio

Verificare che in un albero (che per definizione è sempre un grafo orientato) non esistono cicli.

Percorso in un grafo stradale (senza cicli)

In questo paragrafo viene discusso il problema di trovare un percorso in un grafo.

Ricordando le precedenti scomposizioni scritte per trovare percorsi tra due città costituiti da uno, due o tre tronchi stradali

```
percorso1-2-3(X,Y,[X,Y]) :- tr1(X,Y).
percorso1-2-3(X, Y, L) :- percorso2(X, Y, L).
percorso1-2-3(X, Y, L) :- percorso3(X, Y, L).

percorso2(X,Z,Y) :- tr1(X,Z), tr1(Z,Y).
percorso3(X,Z,W,Y) :- tr1(X,Z), tr1(Z,W), tr1(W,Y).
```

è possibile definire la seguente (prima) relazione di «raggiungibilità»

```
raggiungibili1(X,Y) :- percorso1-2-3(X,Y,_).
```

Questa definizione di raggiungibilità è posta come problema decisionale e non è richiesto di fornire esplicitamente la lista che definisce il percorso. Con questa scomposizione, due città X e Y vengono definite raggiungibili se esiste (almeno) un percorso formato da 1, 2 o 3 tronchi stradali che le congiunge.

Viene ora introdotta una diversa definizione di raggiungibilità che sia a priori indipendente dal numero di tronchi intermedi. Si consideri il seguente programma.

```
raggiungibili(A,Z) :- arco(A,Z).
raggiungibili(A,Z) :- arco(A,B), raggiungibili(B,Z).
```

Naturalmente questo programma si legge nella maniera seguente:

- due città A e Z sono raggiungibili se esiste un arco stradale fra A e Z, oppure
- due città A e Z sono raggiungibili se esiste un arco stradale tra A e la città B e B e Z sono raggiungibili.

Con il programma formato dalla definizione di raggiungibilità sopra data e dalla tabella `{arco(Città1,Città2)}` definita nel precedente paragrafo, sono possibili le seguenti interazioni (immediatamente comprensibili se si disegna il grafo corrispondente a questa tabella).

```
?- raggiungibili(a,h).

SI

?- raggiungibili(h,a).

NO

?- raggiungibili(b,x).

NO

?- raggiungibili(b,X).

X = c;

X = d;

X = i;

X = h;

NO
```

Il grafo associato alla tabella ha una particolarità: è un albero (in particolare è un grafo senza cicli). Per estendere la definizione di raggiungibilità al caso più generale (ma estremamente comune) di grafi non orientati con cicli (corrispondente per esempio alla rete stradale di una qualsiasi regione) è utile estendere o generalizzare il concetto di raggiungibilità relativo a due nodi aggiungendo la richieste di produrre il percorso completo fra questi. In questo caso la formalizzazione del problema contiene tre entità (il nodo di partenza, quello d'arrivo e la lista dei nodi che definiscono il percorso) ed è la seguente

```
{raggiungibili1(Nodo1,Nodo2, Percorso)}.
```

Alcuni casi di prova relativi alla tabella `{arco(Città1,Città2)}` vista sono i seguenti

```
{?- raggiungibili1(a,e,[e,a])}.

{?- raggiungibili1(a,i,[i,d,b,a])}.
```

Il percorso tra due nodi può essere descritto indifferentemente dal primo all'ultimo o dall'ultimo al primo: la scelta qui fatta è spiegata successivamente. Il programma che costruisce (passo per passo) la lista che descrive il percorso può essere costruito facendo riferimento a una scomposizione che preveda sempre (almeno) due alternative

- l’aggiornamento della lista di mano in mano che il percorso procede e
- il criterio d’arresto che deve interrompere il processo di aggiornamento quando è stato raggiunto il nodo finale.

Inoltre, poiché il risultato si ottiene per passi successivi (in questo caso l’aggiunta di un singolo nodo che aggiorna il percorso fino a quel momento trovato), si deve inizialmente stabilire anche il valore iniziale con cui far partire il processo di aggiornamento; nel caso specifico il valore iniziale per la lista (prima di iniziare il percorso) deve essere la lista contenente solo la città di partenza. Se la formalizzazione del problema è la seguente

```
{raggiungibili1(Nodo1,Nodo2, Percorso)}
```

la scomposizione complessiva del problema si scrive quindi con le seguenti tre clausole

```
raggiungibili1(A,Z,L) :- rag(A,Z,L,[A]).
rag(A,Z,L,P) :- arco(A,B), rag(B,Z,L,[B|P]).
rag(Z,Z,L,L).
```

Con la prima clausola, il problema `raggiungibili1(A,Z,L)` viene riscritto, cioè sostituito, con il problema `{rag(A,Z,L,[A])}`, dove il quarto argomento dispone che la prima approssimazione del percorso è costituita appunto dalla lista `[A]`.

La seconda clausola provvede ad eseguire gli aggiornamenti: ad ogni passo ci si sposta di un arco, il percorso viene allungato con un nuovo nodo e si continua a partire da questo nuovo nodo, ripetendo l’aggiornamento finché questo nuovo nodo non è uguale a quello finale: poiché gli elementi vengono aggiunti alla lista da sinistra, l’ordine in cui viene prodotto il percorso (dall’ultimo nodo al primo) è inverso a quello naturale (dal primo all’ultimo).

La terza clausola specifica il criterio d’arresto: quando il percorso aggiornato conduce al nodo finale, viene assegnato al terzo argomento (che rappresenta il percorso finale) il medesimo valore in quel momento posseduto dal quarto argomento (il risultato dell’ultimo aggiornamento).

Alcune interazioni con questo programma e con la tabella già vista `{arco(Città1,Città2)}` sono le seguenti.

```
?- raggiungibili1(a,h,P).
P = [h,d,b,a];
NO
?- raggiungibili1(h,a,P).
```

```
NO
?- raggiungibili1(b,x,P).
NO
?- raggiungibili1(b,X,P).
X = c, P = [c, b] ;
X = i, P = [i, d, b] ;
X = h, P = [h, d, b] ;
X = d, P = [d, b] ;
X = b, P = [b] ;
NO.
```

Esercizio

Il lettore è invitato a individuare il procedimento svolto dal programma per costruire le diverse risposte (in particolare giustificare l'ultima).

Percorso in un grafo stradale (con cicli)

Si consideri ora una tabella che descrive percorsi stradali (a senso unico!) tra due città la cui descrizione implicita sia

```
{arco1(Città1,Città2)}
```

e la cui descrizione esplicita sia

```
arco1(f,g).
arco1(a,f).
arco1(a,b).
arco1(c,b).
arco1(b,d).
arco1(d,c).
arco1(d,i).
arco1(d,h).
arco1(a,e).
arco1(i,m).
arco1(i,n).
arco1(m,p).
```

Grafo con tutti i nodi dell'esempio

Nel disegno è riportata in modo grafico la rappresentazione dell'informazione che nel testo è descritta dalla tabella {arco1(Città1,Città2)}. Anche in questo caso, la direzione è indicata dalla direzione delle frecce, orientato dal nodo di partenza (il primo) al nodo di arrivo (il secondo). Dal grafico risulta evidente il ciclo rappresentato dal percorso che congiunge i nodi b,d,c,d. E' la presenza di questo ciclo che richiede particolari attenzioni nella ricerca di un percorso fra ud enodi. Negli alberi ci sono cicli; nei grafi è sempre possibile che ce ne siano.

Se si risolve il problema (analogo al precedente) di trovare un percorso tra due nodi del grafo rappresentato in questo caso dalla tabella `{arco1(Città1,Città2)}` utilizzando il seguente programma (che differisce dal precedente solo perché usa la tabella `arco1` invece di `arco`)

```
raggiungibili2(A,Z,L) :- rag1(A,Z,L,[A]).

rag1(A,Z,L,P) :- arco1(A,B), rag1(B,Z,L,[B|P]).

rag1(A,A,L,L).
```

si ottengono le seguenti interazioni.

```
?- raggiungibili2(h,a,P).
NO
?- raggiungibili2(i,X,P).
X = p, P = [p, m, i] ;
X = m, P = [m, i] ;
X = n, P = [n, i] ;
X = i, P = [i] ;
NO
?- raggiungibili2(a,h,P).
< messaggio di errore>
```

Il messaggio d'errore che compare si giustifica col fatto che sul percorso fra i nodi `a` e `h` è presente un ciclo (rappresentato dal percorso `[b,d,c,b]` che risulta evidente se si disegna il grafo orientato (associato alla tabella `{arco1(Città1,Città2)}`) con gli archi rappresentati da frecce uscenti dal primo nodo e puntate verso il secondo.

Per evitare questo errore (e quindi impedire al sistema di allungare indefinitamente il percorso girando nel ciclo) si deve introdurre nel programma un controllo per impedire (nella fase di aggiornamento del percorso) di passare per un nodo che sia già presente nella lista che descrive il percorso parziale fatto. Ricordando che il problema di verificare l'appartenenza di un elemento ad una lista è già stato risolto nel paragrafo 6.2, la scomposizione completa del problema in oggetto è la seguente.

```
raggiungibili3(A,Z,L) :- rag2(A,Z,L,[A]).
rag2(A,A,L,L).
rag2(A,Z,L,P) :-
   arco1(A,B),
   not(appartenenza(B,P)),
   rag2(B,Z,L,[B|P]).
```

La prima clausola rappresenta la riscrittura del problema con l'inizializzazione della lista del percorso da costruire mediante aggiornamenti successivi.

La seconda descrive il predicato d'arresto.

La terza clausola rappresenta la scomposizione del ciclo di aggiornamento:

- il primo sottoproblema del ciclo di aggiornamento ricerca un arco che parte dal nodo corrente (in questo caso rappresentato da `A`);
- il secondo sottoproblema verifica che il nodo finale di questo arco (in questo caso `B`) non sia già presente nel percorso parziale fino a quel momento fatto (se la verifica dà risultato positivo, cioè il nodo è già presente sul percorso, il sistema autonomamente cerca un altro arco con origine in A);
- il terzo sottoproblema aggiunge il nuovo nodo al percorso corrente e continua l'aggiornamento del percorso a partire da questo nodo.

Utilizzando questo programma si ottengono le seguenti interazioni.

```
?- raggiungibili3(a,h,P).

P = [h, d, b, a] ;

NO.

?- raggiungibili3(d,X,P).

X = d, P = [d] ;

X = c, P = [c, d] ;

X = b, P = [b, c, d] ;

X = I, P = [i, d] ;

X = m, P = [m, i, d] ;

X = p, P = [p, m, i, d] ;

X = n, P = [n, i, d] ;

X = h, P = [h, d] ;

NO.
```

Esercizio

Scrivere un programma per trovare percorsi nel grafo rappresentato dalla tabella tr1 introdotta nel paragrafo 6.5 e verificarne il comportamento rispetto ai casi definiti dai percorsi che congiungono le seguenti coppie di nodi: `(mi,pd), (rsm,ve), (rn,al), (vr,fi).`

Esercizio

Verificare che il programma appena scritto (in analogia a quanto illustrato al termine del capitolo precedente) può essere direttamente generalizzato per trovare argomentazioni per giustificare enunciati o procedimenti per risolvere problemi (purché le regole di deduzione abbiano un solo antecedente e le formule siano scrivibili come funzioni di una sola variabile).

Un semplice problem solver (1)

Ci sono problemi che si risolvono applicando una formula. Per esempio, date le misure della base e dell'altezza di un rettangolo, trovarne l'area. Per risolvere problemi di questo tipo, si suppone che esista una formula che, noti i valori di alcuni specifici elementi (come le misure della base e dell'altezza del rettangolo), consente di trovare il risultato (nell'esempio, l'area). In questi casi, trovare il procedimento per risolvere un problema consiste nel trovare la formula da applicare.

Le formule in generale prevedono le misure di un numero variabile di elementi: per esempio la formula per calcolare l'area di un trapezio ne prevede tre (le due basi e l'altezza), la formula per calcolare il peso lordo ne prevede due (il peso netto e la tara) e la formula per calcolare l'area di un cerchio ne prevede uno (il raggio).

Per trattare il caso generale di formule con diversi parametri, la struttura implicita della tabella che rappresenta le formule può essere la seguente

{f (N, R, L)}

dove,

N, è il numero identificativo della formula,

R, è il risultato (per esempio l'area del trapezio),

L, è la lista dei parametri (base minore, base maggiore e altezza).

Si sottolinea il forte potere espressivo di questa struttura implicita: qualsiasi formula, con un qualsivoglia numero di elementi, può essere rappresentata da questa struttura. Se, per esempio, si usano lettere minuscole per rappresentare gli elementi coinvolti nelle singole formule, la seguente tabella esplicita rappresenta un insieme di 7 formule che possono essere inserite nel sistema.

```
f(1,a,[b,c]).

f(2,a,[c,d,e]).
```

```
f(3,b,[d,g]).

f(4,e,[d,g]).

f(5,c,[d,g]).

f(6,d,[f,g,h,i]).

f(7,g,[d]).
```

La seconda formula potrebbe rappresentare quella per calcolare l'area del trapezio (chiamata a in questa rappresentazione) e l'ultima quella per calcolare l'area del cerchio (chiamata g, noto il raggio d).

Il problema di trovare la formula per calcolare il valore di un certo elemento, noti i valori degli elementi contenuti in un data lista, si risolve con una semplice interrogazione della tabella. In particolare, il problema di trovare la formula per calcolare b, noti i valori di d,g, si risolve con la seguente domanda

```
?- f(X,b,[d,g]).
```

Con la ovvia risposta `X = 3`.

Esercizio

Discutere il comportamento del sistema rispetto alla seguente domanda.

```
?- f(X,b,[g,d]).
```

Esercizio

Verificare che la seguente regola risolve l'anomalia riscontrata nell'esercizio precedente

```
p(X,Y,L) :- f(X,Y,M), uguali(L,M).
```

Questa regola formalizza il seguente problema : « trovare, se esiste, una regola X per calcolare Y a partire da premesse M che siano tutte e sole quelle assegnate L.

In questa regola, il termine uguali(L,M) rappresenta il sottoproblema di verificare che le due liste L e M rappresentano il medesimo insieme. Verificare quindi che alle due seguenti domande

```
?- p(X,b,[g,d]).

?- p(X,b,[d,g]).
```

il sistema fornisce la medesima risposta. (Hint: due insiemi sono uguali se …)

Esercizio

Verificare che il seguente programma risolve il sottoproblema dell'uguaglianza di due insiemi formalizzato nell'esercizio precedente come {uguali(L,M)} (cioè verificare che il programma sotto riportato risponde SI se e solo se i due insiemi L e M hanno i medesimi elementi).

```
uguali([],[]).

uguali([T|C],M) :- delmem(T,M,N), uguali(C,N) .
```

In questa regola, il termine `delmem(T,M,N)` rappresenta il sottoproblema di togliere dall'insieme M l'elemento T e chiamare N l'insieme rimasto. Con questa scomposizione del problema dell'uguaglianza (di due insiemi), la verifica in questione ha esito positivo se e solo se il secondo insieme contiene tutti e soli gli elementi del primo insieme.

Esercizio

Verificare che il seguente programma risolve il sottoproblema di togliere un elemento da un insieme e restituire il resto.

```
delmem(T,[T|N],N).

delmem(T,[K|M],[K|N]) :- delmem(T,M,N).
```

Esercizio

Ipotizzare le interazioni complete col sistema conseguenti alle seguenti domande

```
?- delmem(X,[a,b,c],Y).

?- delmem(a,[a,b,c,X],Y).

?- delmem(a,[X,b,c,a],Y).

?- uguali([a,b],X).

?- uguali(X,[a,b]).
```

Un semplice problem solver (2)

Data una tabella di formule {f (N, R, L)} la regola

```
p(X,Y,L) :- f(X,Y,M), uguali(L,M).
```

consente di trovare la formula X per calcolare Y assegnati i valori dei parametri contenuti nella lista M. In pratica questa regola individua il procedimento per risolvere un problema primitivo; il problema si risolve infatti applicando la formula individuata dal valore assegnato ad X!

Per problemi non primitivi, il procedimento di soluzione deve contenere almeno due formule, ciascuna corrispondente alla soluzione di un sottoproblema generato dalla scomposizione del problema dato. Per esempio, la regola seguente

```
p1([N3,N2,N1],z,[a,b]) :-
     p(N1,z,[c,d]), p(N2,c,[a,b]), p(N3,d,[a,b]).
```

consente di trovare un procedimento per calcolare z, noti a e b; di fatto la regola implementa la scomposizione del problema in tre sottoproblemi: il tentativo avrà successo se nella base di conoscenza (cioè nella tabella) esistono tre regole, la prima per calcolare z assegnati c e d, la seconda per calcolare c assegnati a e b, la terza per calcolare d assegnati a e b.

Una formula alternativa per risolvere il medesimo problema è la seguente

```
p2([N3,N2,N1],z,[a,b]) :-
    p(N3,d,[a,b]), p(N2,c,[a,b]), p(N1,z,[a,c]).
```

Esercizio

Discutere le due formule nel contesto dei metodi top down e bottom up illustrati nel capitolo 2.

Nel seguito, per trovare procedimenti di soluzione di problemi, si adotterà sistematicamente il metodo top down. Si osserva inoltre che le scomposizioni sopra riportate non sono accettabili perché non si riferiscono a una formulazione generica del problema, ma ad una istanza specifica (trovare z noti a e b). La regola che rappresenta la scomposizione del problema formulato in modo generico è la seguente

```
p3([N3,N2,N1],Z,[A,B]) :-
     p(N1,Z,[C,D]), p(N2,C,[A,B]), p(N3,D,[A,B]).
```

Utilizzando questa regola con la rappresentazione esplicita della tabella {f (N, R, L)} e con le regole illustrate negli esercizi (per i sottoproblemi rappresentati dai termini uguali(L,M) e delmem(T,L,R) si ottiene un programma il cui comportamento è illustrato dalla seguente interazione.

```
?- p3(L,a,[d,g]).

L = [5, 3, 1] ;

L = [3, 5, 1] ;

no
```

Esercizio

Giustificare la soluzione multipla presente nella interazione precedente.

Un semplice problem solver (3)

Le formule riportate nella seguente tabella

```
f(1,a,[b,c]).

f(2,a,[c,d,e]).

f(3,b,[d,g]).

f(4,e,[d,g]).

f(5,c,[d,g]).

f(6,d,[f,g,h,i]).

f(7,g,[d]).

f(8,b,[c,d,e]).

f(9,b,[d,g]).

f(10,e,[a,f]).

f(11,c,[a,f]).

f(12,d,[f,a]).

f(13,g,[d]).
```

possono essere rappresentate da alberi aventi come radice l'elemento che può essere calcolato con quella formula e come foglie gli elementi noti. A titolo di esempio, nella figura seguente sono riportati gli alberi corrispondenti alle formule 1, 3, 5, 6 e 7.

Di conseguenza, anche un procedimento risolutivo può essere rappresentato da un albero; in particolare, i due procedimenti riportati nella interazione precedente (relativa alla soluzione della istanza ?-p3(L,a,[d,g])) sono rappresentate come alberi nella figura seguente

Gli alberi riportati nella figura seguente rappresentano tutti una medesima tipologia di procedimento risolutivo e quindi sono ottenibili con un medesimo programma. Il metodo per ottenerli è descritto da una scomposizione del problema in due sottoproblemi: il primo, f(N,Z,P),

trova una formula N per calcolare il risultato Z (indipendentemente dai parametri presenti nella lista P) e il secondo, pp(L,[N], P, M), trova per ciascuno di questi parametri una formula che consente di determinarli a partire dai dati assegnati; la lista delle formule che costituiscono il procedimento viene costruita per passi successivi e contiene inizialmente la regola individuata dal sottoproblema precedente (vedi secondo argomento).

Indicando con L il procedimento cercato (cioè la lista delle formule da applicare) e con M la lista dei dati, il programma in questione è il seguente.

```
p4(L,Z,M) :-  f(N,Z,P), pp(L,[N], P, M).

pp(L,Q,[T|C],M) :- p(N,T,M), pp(L,[N|Q],C,M).

pp(L,L,[],M).
```

Esercizio

Verificare il comportamento del programma p4, utilizzando la tabella sopra riportata contenete 13 formule, formulando domande per risolvere i seguenti problemi.

Trovare un procedimento (cioè la lista delle formule da applicare) per calcolare il valore di b, conoscendo i valori di a e di f.

Un problem solver generalizzato

Con l’introduzione della scomposizione descritta dalla regola p4 si ottiene una evidente generalizzazione del problema; tuttavia questa generalizzazione è insufficiente perché riguarda solo l’espansione dell’albero in larghezza (col vincolo ulteriore che ciascun parametro contenuto nella

lista P della regola fondamentale, f(N,Z,P) deve essere deducibile dai medesimi antecedenti (rappresentati dagli elementi contenuti nella lista M). In particolare, il programma generalizzato per trovare procedimenti risolutivi di problemi, assemblando formule rappresentate col termine {f(N,Z,L)}, deve poter risolvere problemi con alberi soluzione come quelli riportati a titolo di esempio nella seguente figura; in questa figura, i numeri inseriti tra i rami degli alberi elementari rappresentano il numero della regola corrispondente.

Dall'esame dell'albero, si deduce facilmente che esso rappresenta un procedimento (descritto anche dalla lista [3, 2, 4, 1]) per calcolare x1 conoscendo x4 e x5; l'albero fornisce una visione spaziale del procedimento, mentre la lista ne rappresenta una visione sequenziale specificando l'ordine in cui le formule devono essere applicate. I nodi x2 e x3 sono foglie (pur non essendo dati iniziali del problema) perché quando si deve calcolare x6 con la formula 4, i valori di x2 e x3 sono già stati calcolati in precedenza utilizzando le formule 2 e 3. Una ultima, importante, osservazione è la seguente: su nessuno dei rami che collegano la radice alle foglie devono esserci nodi ripetuti (se ci fossero, quel ramo sarebbe infinito!). Per tener conto di queste osservazioni, nella riscrittura del programma si devono introdurre le entità seguenti:

- la lista delle incognite generate con l'espansione in profondità dell'albero (inizialmente contenente solo la incognita del problema),

- la lista dei dati intermedi che saranno via via calcolati durante l'applicazione del procedimento (inizialmente uguale alla lista dei dati),
- la lista delle formule che vanno a comporre il procedimento (inizialmente uguale alla lista vuota

Il programma cercato per trovare la lista L delle formule da applicare, per calcolare l'incognita Z a partire dalla lista D dei dati, è il seguente

```
problem-solver(Z,D,L) :- ps1([Z], D, D2, [], L).
ps1([Z|C], D, D , L, L) :- member(Z,D).
ps1([Z|C], D1, [Z|D2], L1, [N|L2]) :-
          f(N, Z, P),
          disgiunti(P,C),
          ps2(P, [Z|C], D1, D2, L1, L2).
ps2([T|K],Inc,D1,D2,L1,L2) :-
          ps1(T|Inc],D1,Dx, L1, Lx),
          ps2(K,Inc,Dx,D2,Lx,L2).
ps2([], _, D,D,L,L).
member(X,[X|_]).
 member(X,[_|C]) :- member(X,C).
disgiunti([],_).
disgiunti([T|C],M) :- not(member(T,M), disgiunti(C,M).
```

Se si introduce nel sistema questo programma con l'aggiunta della seguente tabella di formule

```
f(1, x1, [x2,x6]).
f(2, x2, [x4,x3,x5]).
f(1, x3, [x4,x5]).
f(1, x6, [x2,x3]).
f(1, x1, [x2,x4]).
```

```
f(1, x5, [x2,x3]).

f(1, x6, [x5,x7]).

f(1, x2, [x5,x7]).
```

Si ottiene un programma che produce le seguenti interazioni, corrispondenti alla soluzione di istanze di problemi rappresentate dalle relative domande. La presenza della chiamata ricorsiva nella scomposizione del problema ps1 produce la sequenza delle regole in ordine inverso rispetto a quello fornito dai programmi precedenti: ora l'ultima regola inserita nella lista (cioè la testa della lista che rappresenta il risultato) è l'ultima a dover essere applicata.

```
?- problem-solver(x1,[x4,x5],L).

L = [1, 4, 2, 3] ;

L = [1, 4, 3, 2, 3] ;

L = [1, 4, 2, 2, 3] ;

L = [1, 4, 3, 2, 2, 3] ;

L = [1, 4, 2, 3, 2, 3] ;

L = [1, 4, 2, 3, 2, 3] ;

L = [1, 4, 3, 2, 3, 2, 3] ;

L = [5, 2, 3] ;

no


?- problem-solver(x1,[x5,x7],L).

L = [1, 7, 8] ;

no
```

Questo programma se usato con una tabella che rappresenta regole di deduzione fornisce come risultato uno schema di ragionamento che giustifica una asserzione date alcune ipotesi. Le regole di deduzione devono essere nella forma di frasi condizionali con una sola asserzione conseguente e un numero qualsiasi di asserzioni antecedenti. Come già visto per le formule, ogni asserzione deve essere rappresentata da una stringa costante

Glossario

Algoritmo

In altri termini, l'algoritmo può essere definito come **la descrizione** di un procedimento che ha le seguenti proprietà: deve essere esplicita e non ambigua per l'interlocutore cui è destinata e il procedimento attivato di volta in volta seguendo il testo della descrizione deve terminare in un tempo finito. Il concetto di algoritmo presuppone quindi sempre la presenza di un linguaggio usato per la descrizione e di due interlocutori (chi produce la descrizione e chi sa eseguire tutte le azioni previste) che lo sanno usare.

La parola algoritmo deriva dal nome del matematico arabo al Khuwarizmi vissuto nel secolo IX. Fino all'avvento del computer, gli algoritmi venivano usati essenzialmente solo in matematica: in questo caso il linguaggio era il gergo matematico e gli interlocutori erano due persone che lo sapevano usare per formulare l'algoritmo e per svolgere le operazioni descritte. Attualmente, il secondo interlocutore è quasi sempre un computer e il linguaggio usato per la descrizione del

procedimento è un linguaggio di programmazione: in questo caso, l'algoritmo (cioè la descrizione del procedimento) assume la forma di un programma.

Calcolabilità

La teoria della calcolabilità si occupa dei criteri che consentono di definire la effettiva risolubilità dei problemi. Ne consegue che un problema si può risolvere in modo effettivo se e solo se esiste un algoritmo per trattare qualsiasi istanza di quel problema. La importanza pratica di questa affermazione sta nel fatto che con i linguaggi di programmazione per computer è possibile descrivere qualsiasi algoritmo.

Complessità

La teoria della complessità definisce una condizione necessaria per applicare una strategia costruttiva per risolvere un problema: l'algoritmo individuato deve essere computazionalmente trattabile. Quando la soluzione di un problema dipende da un parametro n, detto anche dimensione del problema generico (ciò che si verifica per esempio quando l'input è un insieme di n elementi), il problema è trattabile se le risorse (tempo di calcolo e spazio di memoria) crescono in modo polinomiale (e non esponenziale) rispetto alla sua dimensione.

Computer

Nel contesto del problem solving, il computer è un interlocutore capace di eseguire elaborazioni di informazioni descritte utilizzando un linguaggio di programmazione.

Correttezza

La teoria della correttezza si occupa dei problemi e delle metodologie per dimostrare che il comportamento di un programma è coerente con lo scopo per cui è stato scritto. Un programma corretto risolve in modo esatto qualsiasi istanza del problema per cui è stato scritto. Nella pratica è difficile dimostrare la correttezza di un programma; per questo è sempre necessario sottoporre il programma a verifiche (parziali) di correttezza utilizzando una scelta appropriata di casi di prova.

Input

L’input di un problema è rappresentato dall’insieme dei dati noti che definiscono le singole istanze di quel problema che devono essere risolte: per questo motivo spesso si usa la locuzione "dati di input".

Intelligenza Artificiale

Con questo termine, nato nel 1956 durante un convegno al quale parteciparono i padri fondatori di questa disciplina (tra i quali McCarthy, Minsky, Simon e Newell), si fa riferimento al settore dell’Informatica che si occupa della realizzazione di sistemi artificiali capaci di comportamenti che un osservatore comune attribuirebbe all’intelligenza umana; i sistemi artificiali “intelligenti” sono in genere abbastanza complessi e sanno apprendere e adattarsi (con successo) all’ambiente.

Istanza di un problema

Dato un problema generico (per esempio calcolare l’area di un rettangolo), una istanza specifica è rappresentata da un particolare problema di cui si chiede la soluzione (per esempio trovare l’area di un rettangolo con base uguale a 3 cm e altezza uguale a 5 cm). Quindi una istanza specifica di un problema viene definita quando sono noti tutti i dati di input.

Linguaggio di programmazione

E’ un linguaggio artificiale con capacità espressive definite nell’ambito della teoria della calcolabilità (il cui rispetto garantisce che con linguaggi di questo tipo è possibile descrivere qualsiasi algoritmo). I linguaggi di programmazione vengono usati per descrivere (facilmente) procedimenti di elaborazione che devono essere eseguiti da un computer. La nascita dei linguaggi di programmazione è successiva alla comparsa dei computer; la loro motivazione principale è stata quella di semplificare l’attività di scrivere programmi per computer senza richiedere la conoscenza delle particolarità hardware del computer.

Lista

La lista è una particolare struttura che consente di rappresentare tutti gli elementi di una collezione di dati (per esempio i nomi di un elenco, gli elementi di un insieme o le componenti di un vettore); viene usata da particolari linguaggi di programmazione per descrivere i dati coinvolti in elaborazioni descritte da un programma.

Output

In opposizione a input (che rappresenta i dati di un problema), l'output rappresenta il risultato della soluzione di una particolare istanza di un problema.

Problema di decisione

I problemi di decisione sono quei problemi il cui output è rappresentato semplicemente da un "si" oppure da un "no". In genere, i problemi di decisione consistono nel verificare se i dati di input soddisfano opportuni criteri.

Problema generico

Un problema generico (in contrapposizione a istanza specifica) è un problema formulato facendo riferimento solo alle entità coinvolte, senza specificare alcun dato di input (per esempio calcolare l'area di un rettangolo conoscendone le misure della base e dell'altezza). La soluzione di un problema generico si ottiene esibendo un procedimento di soluzione; mentre la soluzione di una istanza specifica, assegnati i dati di input, si ottiene producendo i valori dell'output.

Programma

Un programma è la descrizione di un algoritmo eseguita utilizzando un linguaggio di programmazione. In genere, un algoritmo viene descritto come programma quando il processo di elaborazione deve essere svolto da un computer.

Programmazione

La programmazione, in Informatica, è l'attività che viene svolta per ottenere programmi per computer; essa è parte essenziale del problem solving quando si assume che l'interlocutore/esecutore sia un computer (ovvero quando si vuole che il procedimento sia descritto in modo esplicito e non ambiguo e quindi sia eseguibile in modo meccanico dall'esecutore).

Scomposizione

La scomposizione di un problema in sottoproblemi è l'attività centrale del processo di problem solving; mediante la scomposizione, la soluzione di un problema (complesso) viene ricondotta alla soluzione di un elenco di sottoproblemi (più semplici). Se la scomposizione viene descritta utilizzando un linguaggio di programmazione (per esempio il Prolog), essa è interpretabile come programma eseguibile da un computer.

Stringa

Una stringa è una qualsiasi sequenza costruibile con i caratteri disponibili; con i linguaggi di programmazione, i caratteri disponibili sono quelli presenti sulla tastiera collegata al computer. A seconda del significato da attribuire alle stringhe, possono esserci particolari limitazioni: per esempio alcuni tipi di stringhe possono essere costruite solo coi caratteri alfabetici e con le cifre numeriche; altre devono iniziare con una lettera minuscola (o maiuscola) dell'alfabeto. I vincoli sull'uso dei caratteri per costruire i vari tipi di stringhe sono definiti dalla sintassi del linguaggio di programmazione che si deve usare.

Variabile

La variabile in Informatica, è la struttura sintattica usata nei programmi per rappresentare le entità coinvolte nelle elaborazioni. Le variabili di input vengono istanziate (cioè assumono valori) all'inizio del processo di elaborazione, quelle di output durante questo processo. Lo scopo di un programma è quello di trovare i valori delle variabili di output che corrispondono a (o sono desumibili da) i dati di input.

Informatica Generale

Esci

Test in itinere "CAP 1 - Il problem solving algoritmico"

Tentativo 24

1 Risposta:

a. un problema è calcolabile quando almeno una sua istanza è risolubile

b. un problema è calcolabile quando esiste un algoritmo applicabile a qualunque sua istanza

c. un problema è calcolabile se e solo se è effettivamente risolubile con l'utilizzo di un computer.

d. un problema è calcolabile quando è trattabile

Quando un problema è calcolabile?

2 Risposta:

a. un procedimento che può essere definito senza introdurre un linguaggio formale

b. un procedimento applicabile solo a problemi di tipo matematico

c. un procedimento effettivo utilizzabile per risolvere qualsiasi istanza di quel problema

d. un procedimento di elaborazione utile solo se si usa un computer

Un algoritmo è

3 Risposta:

a. quando è un problema che può essere risolto in generale
b. quando non è stato assegnato il valore dell'input
c. quando riguarda i generi grammaticali
d. quando non è stato assegnato il valore dell'output

Quando si dice che un problema è generico?

4 Risposta:

a. è un caso anomalo di un problema
b. è la formalizzazione di un problema generico
c. è un esempio di soluzione di un problema generico
d. è la formalizzazione di un caso specifico di un problema generico mediante l'assegnazione dell'input

Cos'è una istanza di un problema?

Sei collegato come Silvia Colotti. (Esci)
InfGen

Informatica Generale

Esci

moodle » InfGen » Quiz » Test in itinere "CAP 2 - I metodi top down e bottom up" » Tentativo 7

Test in itinere "CAP 2 - I metodi top down e bottom up"

Tentativo 7

1 Risposta: a. un linguaggio di programmazione può esistere anche se non è disponibile un computer che lo sappia interpretare

b. la capacità espressiva dei linguaggi di alto livello è superiore a quella dei linguaggi macchina

c. la traduzione fra testi scritti in linguaggio di programmazione può essere eseguita da un programma

d. ogni linguaggio di programmazione può essere usato per comunicare ad un qualunque computer

:Quale fra le seguenti affermazioni è falsa:

2 Risposta:

a. la descrizione formale di un problema generico

b. una descrizione di un procedimento che risulti effettivo almeno per una classe di interlocutori

c. una frase sintatticamente corretta scritta in linguaggio naturale

d. il metodo di scomposizione di un problema in sottoproblemi

Quale fra i seguenti costrutti può essere interpretato come un algoritmo

3 Risposta:

a. solo quando i dati del problema sono stringhe

b. solo per particolari istanze del problema

c. quando il problema non è stato ancora formalizzato

d. In linea di principio sempre, purchè il problema sia formulato correttamente

Quando si può applicare il metodo bottom-up?

4 Risposta:

a. la complessità descrive la lunghezza della descrizione di un problema e il numero di istanze che si sanno risolvere

b. la complessità è la teoria che si occupa della esistenza di procedimenti effettivi per la soluzione delle singole istanze di un problema

c. la complessità si occupa di studiare la difficoltà di definire un algoritmo e quante istruzioni sono necessarie per scrivere un programma

d. La complessità si occupa di individuare limiti accettabili per il tempo necessario per risolvere le diverse istanze di un problema

Di quale argomento si occupa la teoria della complessità computazionale?

5 Risposta: a. un computer non sa come comportarsi con frasi scritte o pronunciate in linguaggio naturale

b. un computer non può apprendere dalla sua propria esperienza

c. un computer non sa arricchire la propria conoscenza, senza l'intervento di un programmatore

d. un computer sa svolgere tutti i lavori corrispondenti ad abilità che possono essere apprese per sentito dire

:Con riferimento alle prestazioni di un computer, segnare la frase corretta fra quelle sotto riportate:

6 Risposta:

a. un computer non sa arricchire la propria conoscenza, senza l'intervento di un programmatore

b. la capacità espressiva dei linguaggi di alto livello è inferiore a quella dei linguaggi macchina

c. la traduzione fra testi scritti in linguaggio di programmazione può essere eseguita da un programma

d. un linguaggio di programmazione non può esistere se non è disponibile un computer che lo sappia interpretare

:Quale fra le seguenti affermazioni è vera:

7 Risposta: a. esistono tanti linguaggi di programmazione quanti sono i tipi di computer esistenti

b. la traduzione di testi scritti in un linguaggio di programmazione presenta le medesime difficoltà che si hanno per tradurre un testo scritto in una lingua naturale

c. con un linguaggio di programmazione possono essere descritti solo procedimenti matematici

d. tutto ciò che può essere detto in maniera esplicita e non ambigua, in una qualsiasi lingua naturale, può essere detto anche in un linguaggio di programmazione

:Segnare la frase corretta riportata nel seguente elenco:

8 Risposta:

a. alla formalizzazione di un problema
b. alla definizione dell'input
c. alla formulazione di un algoritmo
d. alla soluzione di singole istanze del problema

A cosa si applica il metodo Top-Down?

9 Risposta:

a. le frasi condizionali possono essere usate per scomporre problemi in sottoproblemi
b. i linguaggi di programmazione non sono adatti per descrivere algoritmi complicati
c. la decidibilità di un predicato può sempre essere decisa da un programma
d. i linguaggi di alto livello sono adatti per i calcolatori di grandi dimensioni

Quale fra le seguenti affermazioni è vera

10 Risposta: a. un computer sa apprendere dall'esperienza

b. non è vero che tutto ciò che può essere detto in maniera esplicita e non ambigua può essere appreso da un computer

c. per ogni linguaggio di programmazione di basso livello (compreso dalla macchina) esiste un corrispondente linguaggio di alto livello (compreso e usato dai programmatori)

d. esiste un solo linguaggio di programmazione comprensibile da tutti i computer, indipendentemente dal costruttore della macchina

Quale fra le seguenti affermazioni è vera

11 Risposta:

a. è la descrizione di un procedimento finito e non ambiguo per trovare la soluzione di un problema

b. è una procedura per fare dei calcoli matematici

c. è un programma per rappresentare in forma grafica la soluzione di un problema sullo schermo di un computer

d. è la componente della CPU dei computer che consente l'esecuzione automatica dei programmi

Quale fra le seguenti ricorda la definizione di algoritmo?

Informatica Generale

Test in itinere "CAP 3 - La risoluzione dei problemi primitivi"

Tentativo 1

1 Dire quale delle seguenti formalizzazioni è appropriata per il problema «Calcolare l'area di un cerchio, dato il raggio».

Risposta:

a. {superficie(raggio,Area)}.
b. {area-cerchio(Raggio, Area)}.
c. {area(Cerchio,Raggio,Area)}.
d. {area-cerchio(Raggio,Pi-greco,Circonferenza)}.

2 Dato il seguente programma

```
area-rombo(Diagonale1, Diagonale2, Area) :-
  Area = Diagonale1*Diagonale2/2.
```

Trovare la risposta (o le risposte) alla seguente domanda:

```
?- area-rombo(3, 4,Risposta: X).
```

a. No
b. X=6;No
c. Yes
d. X=3;X=6;No

3 Dato il seguente programma

a(b,c,1). a(b,d,2). a(b,c,2).
a(c,d,2). a(c,c,1). a(b,b,2).
a(b,d,1). a(e,d,2). a(e,c,1).
a(b,e,2). a(b,f,1). a(e,f,2).

Trovare la risposta (o le risposte) alla seguente domanda:

?- a(b,f,X).

Risposta:
a. X=c
b. X=1
c. 1
d. No

4 Dire quale delle seguenti formalizzazioni è appropriata per il problema «Dati gli iscritti ad una società sportiva, trovare l'elenco dei praticanti uno sport assegnato».

Risposta:
a. {praticanti(Soci, Sport, Lista)}.
b. {sport(Sport,Società,Elenco,Soci)}.
c. {società(Praticanti,Sport)}.
d. {soci(sport,Praticanti,Lista)}.

5 Dato il seguente programma

a(b,c,1). a(b,d,2). a(b,c,2).
a(c,d,2). a(c,c,1). a(b,b,2).
a(b,d,1). a(e,d,2). a(e,c,1).
a(b,e,2). a(b,f,1). a(e,f,2).

Trovare la risposta (o le risposte) alla seguente domanda:

?- a(X,f,1). Risposta:

a. X=a;No
b. X=b;No
c. b;No
d. X=b;X=a;No

6 Dato il seguente programma

p(A,B,C) :- C = A + B**2.

Trovare la risposta (o le risposte) alla seguente domanda:

?- p(5,1,Q). Risposta:

a. Q=6;No
b. Q=5;No
c. 6;No
d. No

7 Dato il seguente programma

a(b,c,1). a(b,d,2). a(b,c,2).
a(c,d,2). a(c,c,1). a(b,b,2).
a(b,d,1). a(e,d,2). a(e,c,1).
a(b,e,2). a(b,f,1). a(e,f,2).

Trovare la risposta (o le risposte) alla seguente domanda:

?- a(b,X,1). Risposta:

a. Yes
b. X=d; X=f; No
c. X=c; X=d; X=f; No
d. X=c; X=e; X=f; No

8 Dato il seguente programma

a(b,c,1). a(b,d,2). a(b,c,2).
a(c,d,2). a(c,c,1). a(b,b,2).
a(b,d,1). a(e,d,2). a(e,c,1).
a(b,e,2). a(b,f,1). a(e,f,2).

Trovare la risposta (o le risposte) alla seguente domanda:

?- a(X,X,2). Risposta:

a. X=a; X=b; No
b. Yes
c. X=b; No
d. No

9 Dato il seguente programma

a(b,c,1). a(b,d,2). a(b,c,2).
a(c,d,2). a(c,c,1). a(b,b,2).
a(b,d,1). a(e,d,2). a(e,c,1).
a(b,e,2). a(b,f,1). a(e,f,2).

Trovare la risposta (o le risposte) alla seguente domanda:

?- a(e,c,1).

Risposta:
a. Yes
b. e=c
c. 1
d. No

10 Dire quale delle seguenti formalizzazioni è appropriata per il problema «Dato un insieme di numeri naturali, trovare il più grande».

Risposta:
a. {max(Numeri,Naturali,Massimo)}.
b. {max(massimo,naturali)}.
c. {massimo(Insieme, Max)}.
d. {massimo(Numeri-naturali).}

11 Dato il seguente programma

q(X,Y,W,Z) :- Z = (X + Y)/(W**2 + 1).

Trovare la risposta (o le risposte) alla seguente domanda:

?- q(3, 7, 2, Z).

Risposta:

a. Z=2; No
b. X=3;Y=7;No
c. Yes
d. X=3,Z=50;No

Sei collegato come Silvia Colotti. (Esci)
InfGen

Informatica Generale

Esci

moodle » InfGen » Quiz » Test in itinere "CAP 4 - Scomposizioni sequenziali e successive" » Tentativo 1

Test in itinere "CAP 4 - Scomposizioni sequenziali e successive"

Tentativo 1

1 Riconoscere quali sono formalizzazioni appropriate del problema

Risposta:

a. {problema1(Città,Nome,Indirizzo,Tel,Professione)}.
b. {città(Nome,Indirizzo,Tel,Farmacisti)}.
c. {torino(Nome,Indirizzo,Tel,Farmacisti)}.
d. {farmacisti(città,Nome,Indirizzo,Tel,Professione)}.

«Trovare nome, indirizzo e telefono dei farmacisti che risiedono a Torino».

2 Riconoscere quali sono formalizzazioni appropriate del problema

Risposta:

a. {nome(Città,Nome,Indirizzo,Tel,Professione)}.

b. {città(Nome,Indirizzo,Tel,avvocati)}.

c. {avvocati(città,Nome,Indirizzo,Tel)}.

d. {nome(città,Nome,Indirizzo,Tel,Avvocati)}.

«Trovarne il nome, la città di residenza e l'indirizzo di un avvocato che ha 333444555 come numero telefonico».

3 Riconoscere quali sono formalizzazioni appropriate del problema

Risposta: a. {avvocati(bologna,Nome,Indirizzo,Tel,Professione)}.
b. {bologna(Nome,Indirizzo,Tel,Avvocati)}.
c. {città(Nome,Indirizzo,Tel,Avvocati)}.
d. {nome(Città,Nome,Indirizzo,Tel,Professione)}.

«Trovare nome, telefono e indirizzo di tutti gli avvocati che esercitano a Bologna».

Test in itinere "CAP 5 - Scomposizioni condizionali"

Tentativo 1

1 Dato il seguente programma

q(X,Y,Z) :- X > 1+Y, Z is X+Y+1.
q(X,Y,Z) :- X = 1+Y, Z is X+Y.
q(X,Y,Z) :- X < 1+Y, Z is X+Y-1.

e posta la domanda:

?- q(3,2,Z).

Scegliere fra le seguenti la risposta che darà il sistema. Risposta:

a. No; Z=6
b. Z=5
c. Z=4
d. Z=6

2 Dato il seguente programma

q(X,Y,W,Z) :- X < 1+W, Z is X+Y-1.
q(X,Y,W,Z) :- X > 1+W, Z is X+Y+1.
q(X,Y,W,Z) :- X = 1+W, Z is X+Y.

e posta la domanda:

?- q(1,2,3,2).

Scegliere fra le seguenti la risposta che darà il sistema. Risposta:

a. W=2
b. Z=2
c. No
d. Yes

3 Dato il seguente programma

q(X,Y,W,Z) :- X > 1+Y+W, Z is X-Y-1.
q(X,Y,W,Z) :- X < 1+Y+W, Z is X-Y+1.
q(X,Y,W,Z) :- X = 1+Y+W, Z is X-Y.

e posta la domanda:

?- q(4,3,2,Z).

Scegliere fra le seguenti la risposta che darà il sistema.

Risposta:

a. Z=7
b. Z=4
c. Z=6
d. Z=2

4 Dato il seguente programma

q(X,Y,Z) :- X > 1+Y, Z is X+Y+1.
q(X,Y,Z) :- X = 1+Y, Z is X+Y.
q(X,Y,Z) :- X < 1+Y, Z is X+Y-1.

e posta la domanda:

?- q(1,2,3).

Scegliere fra le seguenti la risposta che darà il sistema.

Risposta:

a. Z=4
b. No
c. Z=2
d. Z=3

5 Dato il seguente programma

```
q(0, Y, Z) :- Z = Y.
q(X,Y,Z) :- X < 1+Y, Z is X+Y-1.
q(X,Y,Z) :- X > 1+Y, Z is X+Y+1.
q(X,Y,Z) :- X = 1+Y, Z is X+Y.
```

e posta la domanda:

```
?- q(0,3,Z).
```

Scegliere fra le seguenti la risposta che darà il sistema.

Risposta:

a. Z=3
b. Z=1
c. Z = 3; Z = 2
d. Z=2

6 Dato il seguente programma

```
q(X,Y,W,Z) :- X < 1+W, Z is X+Y-1.
q(X,Y,W,Z) :- X > 1+W, Z is X+Y+1.
q(X,Y,W,Z) :- X = 1+W, Z is X+Y.
```

e posta la domanda:

```
?- q(3,3,3,Z).
```

Scegliere fra le seguenti la risposta che darà il sistema.

Risposta:

a. Z=4
b. Z=7
c. Z=5
d. Z=6

Informatica Generale

Test in itinere "CAP 6 - Scomposizioni ricorsive"

Tentativo 1

Scomposizioni ricorsive

1 Il seguente programma p(L,S,0) deve calcolare la somma S degli elementi contenuti in una lista L, aggiornando il valore del terzo parametro

p([T|C],H4,F) :- G is E + F , p(H5,A,H6). p([ ], H7,H8).

Risposta:

a. H4=C , H5=E , H6=F, H7=A
b. H4=A , H5=C , H6=G , H8=H7
c. H4=C , H5=E , H6=F, H7=F
d. H4=C , H5=G , H6=F, H7=G

Quali sostituzioni devono essere fatte per le variabili H1, H2, H3, H4, H5, H6 e H7?

2	Con riferimento al programma “somma(L,S)” che calcola la somma S degli elementi contenuti in una lista L, scegliere la risposta del sistema alla seguente domanda: somma([3],S).	Risposta:	no	a. S = 3 ; b. S = 0 ; c. S = 1 ; d. No
			no	
			yes	

3 Con riferimento alle seguenti 8 regole di deduzione

r(1,a,[b,c])
r(2,a,[c,d])
r(3,b,[d,e])
r(4,b,[d,f])
r(5,c,[f,e])
r(6,c,[d,f])
r(7,c,[d,e])
r(8,d,[e])

Risposta:
a. L'unica giustificazione è data dall'argomentazione [3,7,1]
b. L'unica giustificazione è data dall'argomentazione [7,8,2]
c. Esistono le due argomentazioni [3,7,1] e [7,8,2]
d. Conoscendo solo la coppia e,d non è possibile giustificare a.

scegliere fra le seguenti la risposta che descrive una corretta giustificazione di a nella ipotesi di conoscere la coppia e,d.

4 Se in una lista L=[T|C] si ha T = a e C = [a.b], quali fra le seguenti è la lista in oggetto?

Risposta:
a. L = [a]
b. L = [a,b]
c. L = [a,a,b,b]
d. L = [a,a,b]

5 Con riferimento al grafo descritto dai seguenti 8 archi

a(a,b),
a(b,c),
a(c,d),
a(e,d),
a(a,e),
a(e,c),
a(e,b),
a(b,e)

e al seguente programma

```
p(X,Y,L) :- q(X,Y,L,[Y].
q(X,X,L,L).
q((X,Y,L,P) :- a(W,Y), q(X,W,L,[W|P]).
```

Scegliere fra le seguenti la risposta che fornisce il sistema alla seguente domanda ?- p(b,d,L).

Risposta:
a. L=[b,e,d] ; no
b. L=[b,c,d] ; no
c. L=[b,c,d] ; L=[b,e,d] ; L=[b,e,c,d] ; no
d. L=[b,c,d] ; L=[b,e,d] ; no

6 Il seguente programma p(S,L) deve calcolare la somma S degli elementi contenuti in una lista L

p(H1,H2).

p(H3,[T|C]) :- p(H4,H5), G is H6 + F.

Risposta:

a. H1=H2, H2=H1, H3=G , H4=C , H5=E , H6=F

b. H1=H2 , H2=H1 , H3=T , H4=C , H5=E , H6=F

c. H1=0 , H2=[ ] , H3=G , H4=C , H5=F , H6=T

d. H1=0 , H2=[ ] , H3=T , H4=C , H5=G , H6=F

Quali sostituzioni devono essere fatte per le variabili H1, H2, H3, H4, H5, H6?

7 Quali sono la testa T e la coda C della seguente lista [z,a] ?

Risposta:

a. T = z, C = [a]

b. T = z, C = a

c. T = a, C = [z]

d. T = Z, C = [A]

8 Il seguente programma p(L,S) deve calcolare la somma S degli elementi contenuti in una lista L

p(H1,H2).

p([T|C],H3) :- p(H4,H5), G is H6 + F.

Risposta:
a. H1=[ ] , H2=0 , H3=G , H4=C , H5=F , H6=T
b. H1=0 , H2=[ ] , H3=T , H4=C , H5=G , H6=F
c. H1=H2 , H2=H1 , H3=T , H4=C , H5=E , H6=F
d. H1=H2, H2=H1, H3=G , H4=C , H5=E , H6=F

Quali sostituzioni devono essere fatte per le variabili H1, H2, H3, H4, H5, H6?

9 Con riferimento al programma “appartenenza(T,L)” che verifica l’appartenenza di un elemento T a una lista L, scegliere la risposta del sistema alla seguente domanda:
?- appartenenza(X,[a,b,a,c]).

Risposta:
a. X=a ; X=b ; X=c ; no
b. X=a ; X=b ; X=a ; X=c ; yes
c. X=a ; X=b ; X=a ; yes
d. X=a ; X=b ; X=a ; X=c ; no