ANNO ACCADEMICO 2022/2023

Algoritmi e Strutture Dati

Teoria

Altair's Notes

UNIVERSITÀ DI TORINO

DIPARTIMENTO DI INFORMATICA

Premessa

Licenza

Formato utilizzato

Box di "Concetto sbagliato":

Concetto sbagliato 0.1: Testo del concetto sbagliato

Testo contente il concetto giusto.

Box di "Corollario":

Corollario 0.0.1 Nome del corollario

Testo del corollario. Per corollario si intende una definizione minore, legata a un'altra definizione.

Box di "Definizione":

Definizione 0.0.1: Nome delle definizione

Testo della definizione.

Box di "Domanda":

Domanda 0.1

Testo della domanda. Le domande sono spesso utilizzate per far riflettere sulle definizioni o sui concetti.

Box di "Esempio":

Esempio 0.0.1 (Nome dell'esempio)

Testo dell'esempio. Gli esempi sono tratti dalle slides del corso.

Box di "Note":

Note:-

Testo della nota. Le note sono spesso utilizzate per chiarire concetti o per dare informazioni aggiuntive.

Box di "Osservazioni":

Osservazioni 0.0.1

Testo delle osservazioni. Le osservazioni sono spesso utilizzate per chiarire concetti o per dare informazioni aggiuntive. A differenza delle note le osservazioni sono più specifiche.

1

Introduzione

1.1 Problemi computazionali

Definizione 1.1.1: Problema computazionale

Un ***problema computazionale*** è una *collezione di domande* per cui sia stabilio un criterio per riconoscere le risposte corrette.

Esempio 1.1.1 (Problema computazionale come collezione di domande)

Massimo Comune Divisore

- Ingressi:

  ⇒ Coppie di interi positivi a, b non entrambi nulli.

- Uscite:

  ⇒ Un intero positivo c tale che c divide sia a che b e se un intero positivo d divide a e b allora $d \leqslant c$.

Note:-

Però la definizione sopra è "intuitiva", per cui si necessità di una formalizzazione.

Definizione 1.1.2: Problema computazionale

Un ***problema computazionale*** è una *relazione binaria*, cioè un insieme di coppie ordinate in cui ogni coppia è composta da un ingresso (la domanda) e l'uscita corrispondente (la risposta).

Corollario 1.1.1 Dominio

Il ***dominio*** è l'insieme di istanze che hanno una risposta.
$dom\{R\} = \{i | \exists r.(i, r) \in R\}$

Corollario 1.1.2 Univocità

Un problema computazionale è ***univoco*** se ogni istanza ammette una e una sola soluzione.

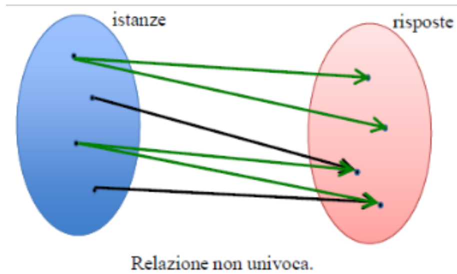

Figure 1.1: Esempio di relazione non univoca.

Esempio 1.1.2 (Problema computaionale come relazione binaria)

Massimo Comune Divisore

$$R = \{((a,b),c) | a \in \mathbb{Z} \wedge b \in \mathbb{Z} \wedge (a > 0 \vee b > 0) \wedge a \mod c == 0 \wedge b \mod c == 0 \wedge$$
$$(\forall d > 0 (a \mod d == 0 \wedge b \mod d == 0) \Rightarrow d \leqslant c)\}$$

1.1.1 Algoritmi

Definizione 1.1.3: Algoritmo

Un ***algoritmo*** è un metodo meccanico per risolvere un problema computazionale.

Corollario 1.1.3 Procedura

Una ***procedura*** è una sequenza finita di operazioni meccanicamente eseguibili, per produrre univocamente un'uscita a partire da certi ingressi.

Note:-

Un algoritmo può essere visto come una procedura che termina per ogni ingresso ammissibile.

Definizione 1.1.4: Determinismo

Un algoritmo è ***deterministico*** se eseguito più volte sullo stesso input fornisce lo stesso output. A ogni algoritmo deterministico è associata una *funzione* dagli ingressi alle uscite.

Corollario 1.1.4 Correttezza

Un algoritmo risolve un problema computazionale R, ossia è ***corretto*** rispetto a R, se, la coppia formata dall'input (generico) e dal corrispondente output è in R.

Note:-

Il primo algoritmo fu proprio quello che calcola il massimo comune divisore, comunemente noto come "Algoritmo di Euclide".

Algoritmo 1.1.1 *(Euclide)*:

```
EUCLID(a, b)        ▷ a > 0 ∨ b > 0
if b = 0 then
    return a
else        ▷ b ≠ 0
    r ← a mod b
    while r ≠ 0 do
        a ← b
        b ← r
        r ← a mod b
    end while
    return b
end if
```

Definizione 1.1.5: Programma

Un ***programma*** può contenere diversi algoritmi. Un programma è scritto in uno specifico linguaggio di programmazione, per cui occorre specificare e implementare diverse ***strutture dati*** $\Longrightarrow$ PROGRAMMA = ALGORITMI + STRUTTURE DATI.

1.1.2 Problemi impossibili e molto difficili

Domanda 1.1

Tutti i problemi computazionali ammettono una soluzione algoritmica?

Risposta: No, esistono problemi *indecidibili*.

Definizione 1.1.6: Halting problem

È possibile sviluppare un algoritmo che, dato un programma e un determinato input finito, stabilisca se il probblema termini o continui la sua esecuzione all'infinito?

Note:-

La risposta è semplicemente ***no***.

Definizione 1.1.7: Problema intrattabile

Un ***problema intrattabile*** è un problema per il quale non esiste un algoritmo con complessità polimoniale in grado di risolverlo.

Corollario 1.1.5 NP-completezza

Classi di problemi tali che tutti o nessuno ammettono una soluzione in tempo polinomiale.

Note:-

I problemi intrattabili e la questione P=NP vengono affrontati in dettaglio nel corso "Calcolabilità e Complessità".

1.2 Correttezza e Terminazione

1.2.1 Correttezza e Logica di Hoare

Bisogna verificare la **correttezza** degli algoritmi (e dei programmi che li implementano).

Definizione 1.2.1: Correttezza totale

Un algoritmo è ***corretto*** se per ogni input fornisce l'output corretto.

Note:-

Tuttavia questa definizione spesso è troppo "forte".

Definizione 1.2.2: Correttezza parziale

Un algoritmo è **parzialmente corretto** se per ogni input *se termina* fornisce l'output corretto.

Note:-

La correttezza parziale viene molto approfondita nel corso "Metodi Formali dell'Informatica" in cui si va a costruire un verificatore per programmi che terminano (utilizzando la logica di Hoare).

Definizione 1.2.3: Logica di Hoare

La logica di Hoare si compone di:

⇒ Precondizioni: che devono essere vere prima dell'esecuzione dell'algoritmo;

⇒ Postcondizioni: che devono essere vere al termine dell'esecuzione dell'algoritmo.

Esempio 1.2.1 (Logica di Hoare)

Divisione intera

- Precondizioni:

  ⇒ $a \geqslant 0$, $b > 0$ numeri interi.

- Postcondizioni:

  ⇒ Il risultato è un intero q tale che $a = b * q + r$ con $0 \leqslant r < b$.

1.2.2 Induzione

Definizione 1.2.4: Induzione semplice

Una proprietà $P(n)$ che vale per $n = 0$ (passo base) e se vale per n allora vale anche $n+1$ (passo induttivo[a]) vale per ogni $n \geqslant 0$.

$$(P(0) \land (\forall n \in \mathbb{N}.P(n) \implies P(n+1))) \implies \forall n \in \mathbb{N}.P(n)$$

[a] Anche detto passo ricorsivo

Corollario 1.2.1 Generalizzazione dell'induzione semplice

Il passo base può anche essere un generico k diverso da 0.

$$(P(k) \land (\forall n \geqslant k.P(n) \implies P(n+1))) \implies \forall n \geqslant k.P(n)$$

Note:-

Inoltre il passo induttivo può anche avere forma $P(n-1) = P(n)$.

Esempio 1.2.2 (Dimostrazione con induzione semplice)

Dimostriamo che $\sum_{k=1}^{n} = (2k-1) = n^2$

⇒ Passo base: si parte da 1 (il primo valore assegnato a k) e si termina sempre con 1 ($n = 1$)

$$\sum_{k=1}^{1}(2k-1) = 1^2 = 1$$

⇒ Passo induttivo: supponiamo che la proprietà sia vera per n

$$\sum_{k=1}^{n+1}(2k-1) = \sum_{k=1}^{n}(2k-1)) + (2(n+1)-1) = n^2 + 2n + 1 = (n+1)^2$$

Definizione 1.2.5: Induzione completa

Una proprietà $P(n)$ che vale per $n = 0, 1, \dots, k$ (passo base) e se vale per $0, 1, \dots, n$ allora vale per $n+1$ (passo induttivo) vale per ogni $n \geqslant 0$.

$$(P(0) \wedge P(1) \wedge \dots \wedge P(k) \wedge (\forall n \geqslant k.(P(0) \wedge P(1) \wedge \dots \wedge P(n)) \Longrightarrow P(n+1))) \Longrightarrow \forall n \in \mathbb{N}.P(n)$$

Teorema 1.2.1 Teorema sui numeri primi

Ogni numero naturale $n \geqslant 2$ è un numero primo oppure è esprimibile come prodotto di numeri primi.

Dimostrazione: si procede per induzione strutturale su n.

⇒ Passo base: è vero per $n = 2$ perché 2 è un numero primo;

⇒ Passo induttivo: si deve dimostrare che se il teorema è vero per ogni $n' \in \{2, 3, \dots, n\}$ allora è vero per $n+1$:

- se $n+1$ è primo allora l'asserto è banalmente vero;
- se $n+1$ non è primo allora si può esprimere $n+1$ come il prodotto di 2 numeri più piccoli n_1 e n_2 con $1 < n_1 < n$ e $1 < n_2 < n$. Dopodiché si applica ricorsivamente l'induzione su n_1 e su n_2.

1.2.3 Dimostrazione di correttezza di algoritmi iterativi (con invarianti)

Definizione 1.2.6: Invariante di ciclo

L'***invariante di ciclo*** è una proposizione sul valore delle variabili "intorno" a un ciclo:

⇒ *Inizializzazione:* la proposizione vale immediatamente prima di entrare nel ciclo;

⇒ *Mantenimento:* se la proposizione è valida prima di eseguire il corpo del ciclo allora sarà valida anche dopo.

Note:-

Serve a dimostrare la correttezza di algoritmi iterativi.

Definizione 1.2.7: Algoritmo di Hornet

L'***algoritmo di Hornet*** è un algoritmo iterativo per calcolare il valore di un polinomio rappresentato dai suoi coefficienti $a_0, a_1, \dots, a_n$ in un punto x.

Algoritmo 1.2.1 *(Hornet)*:

```
1: HORNER(a_0, a_1, ..., a_n, x)
2: y ← 0
3: i ← n
4: while i ≥ 0 do
5:     y ← a_i + x · y
6:     i ← i − 1
7: end while
8: return y
```

Note:-

Per individuare l'invariante si prendono in considerazione i valori delle variabili alla riga 3 (inizializzazione) e alla riga 6 (mantenimento).

i	y
n	0
$n-1$	a_n
$n-2$	$a_{n-1} + a_n x$
$n-3$	$a_{n-2} + a_{n-1}x + a_n x^2$
$n-4$	$a_{n-3} + a_{n-2}x + a_{n-1}x^2 + a_n x^3$
$\vdots$	$\vdots$

Da ciò si può dedurre che la relazione tra i e y, ossia l'*invariante di ciclo* è:

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k$$

Dimostrazione: con $y = 0$ e $i = n$ l'invariante è soddisfatto (prima di eseguire il ciclo). Per dimostrare il dopo aggiungiamo y' e i' (i valori aggiornati):

$$i' = i - 1, \ \ i = i' + 1, \ \ y' = a_i + xy$$

Così si può scrivere:

$$y' = a_i + x \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k$$

Si porta x dentro la sommatoria:

$$y' = a_i + \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^{k+1}$$

Si introducono $k' = k + 1$ e $k = k' - 1$:

$$y' = a_i + \sum_{k'=1}^{n-i} a_{k'+i} x^{k'}$$

Così si può portare dentro alla sommatoria a_i:

$$y' = \sum_{k'=0}^{n-i} a_{k'+i} x^{k'}$$

Infine:

$$y' = \sum_{k'=0}^{n-(i'+1)} a_{k'+i'+1} x^{k'}$$

Il ché è equivalente a:

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^{k}$$

1.2.4 Accenni di terminazione

Non esiste alcun algoritmo in grado di decidere se data una procedura ed un ingresso la procedura termina (Halting problem). In generale è facile avere delle procedure che non terminino, per esempio esistono algoritmi che terminano se l'input rispetta determinate caratteristiche (e. g. è intero), ma non termina se ne ha altre (e.g. è razionale).

2
La complessità e il tempo di calcolo