Introduzione a problemi e algoritmi

February 28, 2019

Obiettivi: introduzione allo sviluppo ed all'analisi degli algoritmi.
Argomenti: problemi computazionali, algoritmi, insolubilità ed intrattabilità, il problema "Peak finding".

Gli **algoritmi cambiano il mondo**: "I fantastici 9" (un libro) spiega in dettaglio 9 algoritmi che hanno già cambiato il mondo (Indicizzazione nei motori di ricerca, PageRank, Crittografia a chiave pubblica, Codici a correzione d'errore, Riconoscimento di forme, Compressione dei dati, Coerenza nei database, Firme digitali, Verificatore di programmi).

Vista l'importanza, a scuola viene introdotto il cosiddetto **"pensiero computazionale"** (la quarta abilità di base oltre leggere, scrivere e calcolare) pure per bambini piccoli. Knuth: "In realtà una persona non ha davvero capito qualcosa fino a che non è in grado di insegnarla a un computer", cioè fino a che non è in grado di ricavarne un algoritmo (un programma) che risolve il problema in questione.

1 Problemi computazionali

Un **problema computazionale** è una collezione di domande, le istanze, per cui sia stabilito un criterio (astratto) per riconoscere le risposte corrette.

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 $d \leq c$.

Più formalmente: 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). Questa relazione definisce come devono essere gli ingressi, come devono essere le uscite, e, dato un certo ingresso, come deve essere l'uscita per essere la soluzione.

Massimo comune divisore.

$$R = \{((a,b),c) | a \in \mathbb{Z} \ \wedge \ b \in \mathbb{Z} \ \wedge \ (a > 0 \vee b > 0) \ \wedge \\ a \bmod c = 0 \ \wedge \ b \bmod c = 0 \ \wedge \ (\forall d > 0.(a \bmod d = 0 \ \wedge \ b \bmod d = 0) \implies d \leq c)\}$$

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

Un problema computazionale (cioè una relazione binaria) è **univoca** se ogni istanza ammette una sola risposta. (Disegno.)

Esempio. Moltiplicazione fra interi (univoca).
$R = \{((a,b),c) | a \in \mathbb{Z} \wedge b \in \mathbb{Z} \wedge a \cdot b = c\}$

Esempio. Fattorizzazione (non univoca).
$R = \{(a,(c_1,...,c_n)) | a \in \mathbb{Z} \ \wedge \ a > 1 \ \wedge \ \prod_{i=1}^{n} c_i = a \ \wedge \ \forall 1 \leq i \leq n.c_i \in \mathbb{P}\}$

Esempio. Fattorizzazione (univoca).
$R = \{(a,(c_1,...,c_n)) | a \in \mathbb{Z} \ \wedge \ a > 1 \ \wedge \ \prod_{i=1}^{n} c_i = a \ \wedge \ \forall 1 \leq i \leq n.c_i \in \mathbb{P} \ \wedge \ \forall 1 \leq i \leq n-1.c_i \leq c_{i+1}\}$

2 Algoritmi

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

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

Un **algoritmo** è una procedura che termina per ogni ingresso ammissibile.

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

Un algoritmo risolve un problema computazionale R, ossia è **corretto** rispetto ad R, se, per qualunque input e l'output corrispondente, la coppia (input, output) è in R.

Certi algoritmi si imparano a scuola: somma in colonna.

Termine algoritmo viene dalla trascrizione latina del nome del matematico persiano **al-Khwarizmi** (780-850) che ha scritto "Algoritmi de numero Indorum" dove descrive operazioni utilizzando il sistema di numerazione indiano.

L'algoritmo considerato il primo algoritmo è quello che calcola il massimo comune divisore, chiamato l'algoritmo di Euclide (367-283 a.C.) ma forse era risaputo già prima:

1: EUCLID(a, b)
2: $r \leftarrow a \bmod b$
3: **while** $r \neq 0$ **do**
4: $\quad a \leftarrow b$
5: $\quad b \leftarrow r$
6: $\quad r \leftarrow a \bmod b$
7: **end while**
8: **return** b

Simulazione dell'algoritmo con $a = 35, b = 55$.

Programma vs. algoritmi.
Un programma può contenere diversi algoritmi. Un programma è scritto in uno specifico linguaggio di programmazione. In un programma occorre specificare ed implementare opportune **strutture dati**.
Programma=Algoritmi+Strutture Dati

3 Problemi impossibili e molto difficili

É vero che tutti i problemi computazionali ammettano una soluzione algoritmica? **No, esistono problemi indecidibili.**

Un famoso problema indecidibile è il **problema della terminazione** che pone la seguente domanda: è possibile sviluppare un algoritmo che, dato un programma e un determinato input finito, stabilisca se il programma in questione termini o continui la sua esecuzione all'infinito. La risposta è no.

Un **problema intrattabile** è un problema per il quale non esiste un algoritmo con complessità polinomiale in grado di risolverlo. Torre di hanoi con n dischi richiede $2^n - 1$ spostamenti.

Esistono problemi, detti **NP-completi**, tali che tutti o nessuno ammettono una soluzione in tempo polinomiale. Esempio: cercare un cammino hamiltoniano (un cammino in un grafo, orientato o non orientato, è detto hamiltoniano se esso tocca tutti i vertici del grafo una e una sola volta).

4 Un esempio per illustrare dei temi che saranno studiati durante il corso

Il problema del "peak finding".
Input: un vettore $A[0...n-1]$ di n numeri interi;
Output: un intero $0 \leq p \leq n-1$ tale che $A[p-1] \leq A[p] \geq A[p+1]$ dove $A[-1] = A[n] = -\infty$.

Esempio: se $A = |1, 2, 6, 6, 5, 3, 3, 3, 7, 4, 5|$ allora abbiamo 5 picchi nelle posizioni 2,3,6,8,10.

Cerchiamo un picco percorrendo il vettore da sinistra a destra fino al picco che si trova più a sinistra.

```
1: PEAK-FIND-LEFT(A[0...n − 1])        ▷ n ≥ 1
2: p ← 0
3: k ← 1
4: while k < n ∧ A[p] < A[k] do
5:     p ← k
6:     k ← k + 1
7: end while
8: return p
```

(Provare ad eliminare la variabile k dall'algoritmo precedente.)

Caratteristiche di questo algoritmo: nel **caso migliore** $p = 0$ è un picco e si fa un singolo confronto (fra gli elementi del vettore); nel **caso peggiore** il picco si trova nell'ultima posizione, si percorre tutto il vettore, e si fanno $n - 1$ confronti. Di solito ci interessa caratterizzare il caso peggiore (oppure il **caso medio** ma qui non ne parliamo).

Nel caso peggiore si percorre tutto il vettore. Con lo stesso sforzo si può trovare il picco più alto:

```
1: PEAK-FIND-MAX(A[0...n − 1])        ▷ n ≥ 1
2: p ← 0
3: for k ← 1 to n − 1  do
4:     if A[p] < A[k] then
5:         p ← k
6:     end if
7: end for
8: return p
```

Questo fatto suggerisce che si può fare meglio.

Cosa garantisce di avere un picco in un certo segmento del vettore $A[i...j]$ con $i \leq j$?

Se $A[i-1] \leq A[i]$ e $A[j] \geq A[j+1]$ allora deve esserci un picco nel segmento $A[i...j]$. Più formalmente:
Teorema:
Siano i e j tali che $0 \leq i \leq j \leq n-1$ e $A[0...n-1]$ un vettore di n interi. Se $A[i-1] \leq A[i]$ e $A[j] \geq A[j+1]$ allora esiste $i \leq p \leq j$ tale che $A[p-1] \leq A[p] \geq A[p+1]$ ossia p è un picco in $A[i...j]$.
Dimostrazione:
Se $i = j$ abbiamo $A[i-1] \leq A[i] \geq A[i+1]$ e quindi $p = i$ è un picco.

Se $i \neq j$ allora scegliamo una qualunque posizione q_1 tale che $i \leq q_1 \leq j$. Abbiamo tre possibilità: $A[q_1-1] \leq A[q_1] \geq A[q_1+1]$ e quindi la posizione q_1 è un picco; $A[q_1-1] > A[q_1]$ e quindi la posizione q_1 non è un picco; $A[q_1] < A[q_1+1]$ e quindi la posizione q_1 non è un picco.

Se q_1 è un picco allora il picco c'è. Se q_1 non è un picco siano $i_1 = i$ e $j_1 = q_1 - 1$ se $A[q_1-1] > A[q_1]$ e $i_1 = q_1 + 1$ e $j_1 = j$ altrimenti.

Si nota che abbiamo $A[i_1 - 1] \leq A[i_1]$ e $A[j_1] \geq A[j_1 + 1]$, cioè i_1 e j_1 soddisfano le condizioni richieste dal teorema ma il segmento $A[i_1...j_1]$ contiene meno elementi del segmento $A[i...j]$.

Ora si ripete lo stesso ragionamento ma sul segmento $A[i_1...j_1]$: se $i_1 = j_1$ allora la posizione $p = i_1$ è un picco; se $i_1 \neq j_1$ allora si sceglie una qualunque posizione q_2 tale che $i_1 \leq q_2 \leq j_1$. Se q_2 è un picco allora un picco c'è. Se non lo è, allora abbiamo $A[q_2 - 1] > A[q_2]$ oppure $A[q_2] < A[q_2 + 1]$. Se q_2 non è un picco siano $i_2 = i_1$ e $j_2 = q_2 - 1$ se $A[q_2 - 1] > A[q_2]$ e $i_2 = q_2 + 1$ e $j_2 = j_1$ altrimenti.

Ora si ripete lo stesso ragionamento ma sul segmento $A[i_2...j_2]$...

Procedendo così o si trova un picco nella sequenza $q_1, q_2, ...$ oppure per una certa k si ha $i_k = j_k$ e quindi i_k è un picco. □
(Una dimostrazione più compatta e meno "procedurale" si ottiene utilizzando l'induzione completa.)

Il teorema, applicato con un vettore $A[0...n-1]$ e $i = 0, j = n-1$ e $A[-1] = A[n] = -\infty$, garantisce la presenza di un picco.

In più la dimostrazione del teorema suggerisce un modo di cercare un picco. Nella dimostrazione si sceglie sempre una posizione arbitraria nel segmento considerato.

Come conviene scegliere la posizione? Con $q_k = \lfloor \frac{i_{k-1}+j_{k-1}}{2} \rfloor$ (siano $i_0 = i$ e $j_0 = j$) in ogni giro si dimezza il segmento considerato. Ne segue un algoritmo di tipo Divide et Impera:

```
1: PEAK-FIND-DI(A[i...j])        ▷ i ≤ j
2: q ← ⌊(i + j)/2⌋
3: if A[q − 1] ≤ A[q] ≥ A[q + 1] then
4:     return p
5: else      ▷ A[q − 1] > A[q] ∨ A[q] < A[q + 1]
6:     if A[q − 1] > A[q] then
7:         return PEAK-FIND-DI(A[i...q − 1])
8:     else
9:         return PEAK-FIND-DI(A[q + 1...j])
10:    end if
11: end if
```

La chiamata PEAK-FIND-DI($A[0...n-1]$) trova il picco nel intero vettore.

(Cosa succede se la scelta è $q_k = i_{k-1}$? Cosa succede se la scelta è $q_k = j_{k-1}$? Simulare l'algoritmo col vettore $A = |1, 2, 6, 6, 5, 3, 3, 3, 7, 4, 5|$.)

Quanti confronti fa l'algoritmo precedente? Per semplicità contiamo solo quante volte viene effettuato il confronto $A[q-1] \leq A[q] \geq A[q+1]$ e lo consideriamo come singolo confronto. Assumiamo di avere $n = 2^k$ per qualche intero k e consideriamo il caso peggiore, cioè in ogni giro il vettore si dimezza esattamente e l'algoritmo termina quando viene effettuata la chiamata con $i = j$.

Il numero di confronti in caso di un vettore di n elementi è

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ T\left(\frac{n}{2}\right) + 1 & \text{se } n > 1 \end{cases}$$

$$\begin{aligned} T(n) &= T\left(\tfrac{n}{2}\right) + 1 \\ &= T\left(\tfrac{n}{4}\right) + 1 + 1 \\ &= T\left(\tfrac{n}{8}\right) + 1 + 1 + 1 \\ &= T\left(\tfrac{n}{2^3}\right) + 3 \\ &= T\left(\tfrac{n}{2^k}\right) + k && \text{per } 1 \leq k \leq \log_2 n \\ &= T(1) + \log_2 n && \text{utilizzando } k = \log_2 n \\ &= 1 + \log_2 n \end{aligned}$$

Quindi, nel caso peggiore, il numero di confronti utilizzando PEAK-FIND-DI è proporzionale a $\log_2 n$ mentre con PEAK-FIND-LEFT è $n-1$. Per esempio, con $n = 2^{20} = 1048576$, $\log_2 n = 20$ mentre $n - 1 = 1048575$.