Il Pensiero Computazionale

Percorso per i Docenti della Scuola Secondaria

Fabrizio Luccio. Algoritmi e Coding

12 Novembre 2018

La parola Algoritmo

Storicamente (sec. XIII): Algorismus è un procedimento di calcolo aritmetico (dal nome del matematico persiano del IX sec. Muḥammad ibn Mūsā al-Khuwārizmī)

Nel sec. XX: correlate al "Problema della decisione" di Hilbert (1928) nascono diverse definizioni formali di algoritmo, fino alla "Macchina di Turing" del 1936

In linguaggio corrente e formalmente ineccepibile algoritmo è una sequenza ordinata e finita di passi elementari per risolvere un problema, che conduce a un risultato in un numero finito di passi

La parola Code (inglese)

Dizionario di Oxford (UK): A system of words, letters, figures, or symbols used to represent others

Dizionario Merriman-Webster (USA): A system of signals or symbols for communication

In informatica: Coding, atto di trasformare un algoritmo secondo un code definito per una macchina. Può essere sostituita con la parola programmare

Problem solving: è l'attività fondamentale su cui si sviluppa il pensiero computazionale. Si articola in diverse fasi

Analisi del problema. Include crucialmente l'esame su come dati e risultati possono essere descritti (struttura dei dati)

Ideazione di un algoritmo di risoluzione, con la previsione dell suo tempo di calcolo (complessità) e il confronto con altri algoritmi possibili

Coding dell'algoritmo scelto, sua analisi di correttezza, e controllo della complessità. Il coding può essere diviso in due livelli: descrizione in pseudocodice, e programmazione nel linguaggio scelto

Un semplice esempio: Ricerca dell'informazione associata a un nome in un elenco.

Il problema è posto per un elenco contenente un numero arbitrario *n* di elementi

La struttura dati naturale è composta di due vettori N e I per contenere nomi e informazioni, i cui elementi hanno indice da 1 a *n* : al nome N[j] corrisponde l'informazione I[j], per $1 \leq j \leq n$.

Il nome da cercare (input) e l'informazione trovata (output) sono contenuti in due variabili.

Immaginiamo che N e I contengano nomi di persone e i rispettivi numeri di telefono. Indichiamo con NOME e TEL le variabili di input/output.

La variabile NOME e gli elementi di N sono rappresentati da una sequenza di caratteri alfabetici.

La variabile TEL gli elementi di I sono rappresentati da un numero.

Se NOME = N[j] avremo TEL = I[j]

Se NOME non appare in N porremo TEL = 0

1. L'elenco non ha alcuna strutturazione particolare

j	1	2	3	.....	n
N	Rossi	Bacci	Gotti		Carli
I	35822	42231	41061	.....	72909

2. L'elenco ha già una struttura coerente con l'operazione di ricerca da eseguire: per esempio i nomi vi appaiono in ordine alfabetico

j	1	2	3	.....	n
N	Bacci	Carli	Gotti	.....	Rossi
I	42231	72909	41061	.....	35822

Nel caso 1 (elenco in ordine qualsiasi) un algoritmo ovvio è scandire il vettore N dall'elemento 1 a *n* confrontando ogni elemento con NOME.

La correttezza dell'algoritmo in questo caso è evidente.

Per calcolarne la complessità, ovvero il tempo che impiega, dobbiamo chiarire alcuni punti.

- Il tempo non si misura in secondi ma in numero di operazioni elementari, ciascuna delle quali richiede ragionevolmente un tempo costante
- Tale numero è stimato in ordine di grandezza come funzione di n
- L'analisi è di caso pessimo, cioè si riferisce alla disposizione iniziale dei dati che richiede il massimo numero di operazioni

Il caso pessimo è quando l'elemento NOME è in ultima posizione, cioè NOME = N[*n*], o non è contenuto in N: sono allora necessari *n* passi di scansione e relativi confronti (operazioni elementari).

La complessità in ordine di grandezza è $O(n)$: l'algoritmo è detto lineare (in tempo).

Possiamo ora passare alla fase di coding programmando l'algoritmo in pseudocodice, un linguaggio che presenta le principali caratteristiche comuni ai linguaggi di programmazione di uso generale ma non richiede vari dettagli tecnici necessari per eseguire questi su un calcolatore.

Nel seguente programma RICERCA ammettiamo che i vettori N e I siano presenti nella memoria del calcolatore (quindi anche *n* è noto).

Chiamiamo il programma comunicando a esso il valore di NOME. Il programma risponde con il valore di TEL.

```
programma RICERCA(input NOME, output TEL)
    j ← 1;
    while j ≤ n
        if N[j] = NOME
            TEL ← I[j];
            stop
        else j ← j+1;
    TEL ← 0;
```

Programma RICERCA-GUIDATA per caricare i dati nella memoria e richiedere al calcolatore i numeri di telefono di uno o più utenti.

Le variabili NOME e TEL non appaiono nell'intestazione e vengono gestite dal programma.

RICERCA-GUIDATA utilizza (chiama) il programma precedente RICERCA al suo interno.

Se la variabile interna CERCA = 1 il programma cerca un nuovo numero, per CERCA = 0 si arresta.

```
programma RICERCA-GUIDATA
    get N;
    get I;
    CERCA ← 1;
    while CERCA = 1
        print "comunica NOME";
        get NOME;
        RICERCA(NOME,TEL);
        print TEL;
        print "continuare la ricerca?"
        get CERCA;
```

Nel caso 2 (elenco dei nomi ordinato in ordine crescente) possiamo applicare l'algoritmo manuale di ricerca in una rubrica, nel nostro esempio l'elenco del telefono: si veda Computational Thinking di Jeanette Wing.

A questo scopo sono necessarie alcune precisazioni.

I nomi nella rubrica sono sequenze di caratteri e si ordinano a mano in ordine alfabetico. Nel calcolatore tali sequenze sono rappresentate in binario e possono essere trattate come numeri binari, dunque è naturale impiegare la relazione aritmetica $\leq$ nel confronto.

In una rubrica gli accessi manuali ai dati sono eseguiti in modo euristico. Nel calcolatore possiamo calcolare in tempo costante un valore dell'indice j (per esempio il valore centrale tra 1 e n) e accedere in tempo costante a N[j].

L'algoritmo, noto come RICERCA-BINARIA, consiste ora nei passi seguenti:

- calcolare il valore centrale m dell'indice j
- confrontare NOME con N[m]: se coincidono, estrarre l'informazione I[m], altrimenti:

  se NOME < N[m] ripetere il punto precedente sul sottovettore di sinistra N[1] . . . N[m-1]

  se NOME > N[m] ripetere il punto precedente sul sottovettore di destra N[m+1] . . . N[n]

  se un sottovettore è vuoto, NOME non appare in N e l'algoritmo si arresta

Anche in questo caso la correttezza dell'algoritmo è evidente.

Quanto al tempo di calcolo, il caso pessimo è quando l'elemento NOME è in ultima posizione nella ricerca o non è contenuto in N.

Questo richiede tante operazioni di scansione e relativi confronti, per quanti sono i passi necessari a ridurre a un singolo elemento il sottovettore su cui si esegue via via la ricerca. Tali passi sono $\approx \log_2 n$.

La complessità in ordine di grandezza è $O(\log n)$

```
programma RIC-BIN(input NOME, output TEL)
    j ← 1; k ← n;
    while j ≤ k
        m ← inf{(j+k)/2};
              // inf{x} è il massimo intero ≤ x
        if NOME = N[m]
            TEL ← I[m];
            stop;
        if NOME < N[m] k ← m-1
        else {// cioè NOME > N[m]} j ← m+1;
    TEL ← 0;
```

Quello che un algoritmo non può fare

$a^n + b^n = c^n$

non ha soluzione intera positiva per $n > 2$

è una famosa affermazione di Fermat dimostrata da Andrew Wiles nel 1995 in più di 130 pagine (ma la sua prima dimostrazione era sbagliata . . .)

Problema: si potrebbe dimostrare che l'affermazione è vera mediante un algoritmo ?

Costruiamo un algoritmo che prova tutti i valori possibili per a, b, c, n con $n > 2$, si arresta se trova una quaterna per cui $a^n + b^n = c^n$, e stampa i valori a, b, c, n

questo ci permetterebbe di dimostrare che l'affermazione di Fermat è falsa e anche la nuova dimostrazione di Wiles è sbagliata

ma devo formulare l'algoritmo in modo che una eventuale quaterna per cui $a^n + b^n = c^n$ venga raggiunta in un numero finito di passi

FERMAT

for (i **from** 6 **to** ∞, *con incremento* 1)

costruisci le quaterne $q_i = \{n,a,b,c\}$

con $n>2$, $a,b,c>0$, $n+a+b+c=i$;

forany q_i

if ($a^n + b^n = c^n$) *stampa* q_i *e termina*

FERMAT termina se e solo se l'affermazione
di Fermat è falsa per una certa quaterna

Se l'affermazione di Fermat è vera l'algoritmo FERMAT non termina (dunque non può essere utilizzato per la dimostrazione).

Ma questa potrebbe consistere nel formulare FERMAT e costruire un altro algoritmo che decida se FERMAT termina o no.

Nel suo famoso articolo del 1936 Turing dimostrò che costruire un simile algoritmo non è possibile !

La conseguenza, legata al problema di Hilbert, è che gli algoritmi non forniscono dimostrazioni gratis

Problemi facili (o trattabili) e difficili (o intrattabili)

Nel mondo degli algoritmi questi termini non sono riferiti alla difficoltà incontrata nel risolvere un problema, ma al tempo di calcolo necessario per tale risoluzione, ovvero alla sua complessità di calcolo

Nei problemi facili possiamo determinare un algoritmo il cui tempo di calcolo è funzione polinomiale nella dimensione dei dati su cui l'algoritmo deve operare.

Nei problemi difficili i migliori algoritmi che siamo capaci di individuare richiedono tempo di calcolo esponenziale nella dimensione dei dati.

Una riflessione sul confronto tra algoritmi di diversa complessità

Un algoritmo di complessità polinomiale risolve un certo problema su n dati in tempo $t = n^s$ su un computer A.

Lo stesso algoritmo, su un computer k volte "più veloce" di A, risolve il problema su $m > n$ dati nello stesso tempo t, dove:

$$n^s = t \quad m^s = kt \text{ , da cui } m^s = kn^s \text{ quindi } m = k^{1/s} n$$

Ripetiamo il ragionamento con un algoritmo di complessità 2^n:

$$2^n = t \quad 2^m = kt \text{ , da cui } 2^m = k2^n \text{ quindi } m = \log_2 k + n$$

Le due principali classi di complessità

P è la classe dei problemi che si risolvono in tempo polinomiale.

NP è la classe dei problemi che si verificano in tempo polinomiale (ma si sanno risolvere solo in tempo esponenziale).

P = NP ?

Per la risoluzione di questo problema è in palio un premio di 1 M $

Tre problemi tra i “più difficili” in NP

Biologia molecolare: date due sequenze S , T contenenti gli stessi elementi in diverso ordine, stabilire se S può essere trasformata in T invertendo non più di k sottosequenze, per k assegnato.

Trasporti: stabilire se un insieme di casse di pesi diversi possono essere caricate su un assegnato numero di camion senza superare il peso massimo sopportabile da ciascun camion.

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

Un campo di applicazione ove la distinzione tra problemi polinomiali e esponenziali è fondamentale è la crittografia, ove si utilizzano funzioni "one-way" che sono facili da calcolare e difficili da invertire.

Il calcolo diretto è richiesto per eseguire e interpretare una comunicazione cifrata, il calcolo dell'inverso è indispensabile per decifrare una comunicazione senza autorizzazione

NEL LABORATORIO VEDREMO

- Approfondimento di nuovi algoritmi
- Per chi ha qualche esperienza di programmazione:
  sviluppo e esecuzione di programmi in linguaggio Python, tra cui quelli per gli algoritmi presentati in questa lezione
- Per chi non ha mai programmato:
  prima esposizione alla programmazione in Python

Ciascuno sarà libero di muoversi tra queste attività dirigendosi su quelle che più gli interessano