Algoritmi e Strutture Dati

Leonardo Marro, Linda Primicino

22 marzo 2023

Indice

I Algoritmi **2**

1 Introduzione **2**

2 Domande e Risposte **2**
2.1 Calcolo del Tempo 2
2.2 Ricerca del "Meglio" 2
2.3 Memorizzazione dei Dati 2

3 Problemi e Algoritmi **3**
3.1 Problemi computazionali 3

4 Algoritmo **3**
4.1 La funzione I/O 3
4.2 Differenza tra Programma e Algoritmo 3

5 Peak Finding **3**

6 Problemi Insolubili **4**
6.1 Problema dell'Halt 4

7 Problemi Intrattabili **4**

8 Analisi Qualitativa **4**
8.0.1 I Disastri Possibili 5
8.1 Verificatore immaginario 5
8.2 Correttezza parziale e totale 5
8.3 Specifica dell'algoritmo 5
8.4 Ricorsione 5
8.4.1 Le torri di Hanoi 5
8.5 Lo schema di induzione 5
8.5.1 Esempio di Ricorsione . . . 6
8.6 Divisione Iterativa 6
8.7 Accumulatori ed Invarianti 7
8.8 Algoritmo di Euclide 7

II Algoritmi **8**

1 Insertion Sort **8**

2 Selection Sort **8**

Parte I

Algoritmi

Testo Teoria, Introduzione agli algoritmi e strutture dati di Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.
Testo Laboratorio (opzionale), Horstmann Concetti di informatica e fondamenti di Java.

1 Introduzione

La vera Informatica si basa sullo sviluppo degli **Algoritmi** caratterizzato da un certo metodo di pensieri:

- Comprendere Problemi;
- Concettualizzare ed astrarre (Algoritmi grafi);
- Cercare soluzioni generali;
- Descriverle con esattezza;
- Valutare la correttezza e l'efficienza delle soluzioni.
  - Le soluzioni possono essere sufficienti o insufficienti, efficienti o inefficienti, esatte o errate.

Grazie a questo metodo di analisi si arriva a diverse realizzazioni che posso usare strutture dati differenti, noi punteremo a sviluppare un modello di calcolo.

Utilizzeremo di sovente dello **pseudo-codice**, ovvero un codice scritto il linguaggio **naturale** non legato ad una particolare scelta, tecnologia, architettura.

Post-Condizione: Descrizione dell'output in base all'input.

Invariante del Ciclo: [missing];

2 Domande e Risposte

Cosa Calcola?	Trovare l'invariante di ciclo
In quanto tempo?	Analisi del Caso peggiore
Come confrontare due algoritmi?	Complessità asintotica
Ci sono soluzioni migliori?	Ricerca del meglio
Come memorizzare efficientemente i dati?	Strutture dati
Come esplorare strutture dati?	Scansione, Ricerca dicotomica e Visita
Quali sono le proprietà dei dati?	Ordinamento, connessione, sotto struttura minima, partizionamenti

2.1 Calcolo del Tempo

Il calcolo del tempo di esecuzione di una funzione si complica notevolmente nel caso di utilizzo di funzioni ricorsive in quanto non siamo a conoscenza del numero di cicli, vedremo più avanti i diversi calcoli e formule necessarie. Esistono due obbiettivi di calcolo: *Analisi del calcolo peggiore*, dove si cerca la durata massima di una funzione, e *Analisi del caso medio*, più complicata siccome usa concetti di probabilità (tempo è una v.a) per trovare la durata del caso più probabile.

2.2 Ricerca del "Meglio"

Durante la creazione degli algoritmi una domanda importante è "Posso fare di meglio?", useremo lo studio di confini superiori e inferiori per la valutazione ed il calcolo della risposta a questa domanda. Questa è una tipica **Domanda di Esame**:

1. Quando x ritorna y?
2. Qual è la complessità in termini O?
3. *Sapreste indicare un algoritmo simile che risponda con un tempo asintoticamente migliora?*

.

2.3 Memorizzazione dei Dati

Un grande problema nello sviluppo è la memorizzazione dei dati, studieremo diverse strutture

dati sia statiche che dinamici e di algoritmi di inserimento che operano su queste ultime. Classificheremo gli algoritmi in due categorie: puramente **inspettive** (non distruttive, non modificano) oppure **distruttivi** (mutiamo i valori) e i relativi criteri. In aggiunta valuteremo la **complessità ammortizzata**, ovvero tutta la vita della struttura dati (anche questo si baserà su una *stima*).

3 Problemi e Algoritmi

3.1 Problemi computazionali

Un problema computazionale è un insieme di domande chiamate **istanze** per cui si ha stabilito un criterio per riconoscere le risposte corrette. Esempi di problemi computazionali:

- Moltiplicazione
- Fattorizzazione
- Ordinamento
- Ordinamento topologico **(R non univoca)**
- Percorso ottimo **(R non univoca)**

Un problema computazionale è una **Relazione** (Attenzione! Non è una funzione!) definibile attraverso l'aritmetica di Peano, ovvero formalizzabile almeno al primo ordine.

$$R = \{[missing]\}$$

$$dom(R) = \{i | \exists r, (i, r) \in R\}$$

NON SEMPRE UNIVOCA, i problemi di computazione possono accettare più risposte.

4 Algoritmo

Un algoritmo è una procedura,che termina per ogni ingresso ammissibile.
Viene definito **corretto** rispetto ad un problema se ad ogni istanza associa un output cui soddisfa il criterio di sicurezza, **un algoritmo corretto risolve un problema computazionale**.
Una procedura è una sequenza finita di operazioni meccanicamente eseguibili.

4.1 La funzione I/O

Prendendo molteplici volte lo stesso input l'algoritmo fornirà sempre la stessa uscita, per questo gli algoritmi sono **deterministici**, si può associare una funzione input-output ad ogni algoritmo.

Un algoritmo è quindi corretto rispetto ad R , se la funzione input/output A associa una risposta ad ogni istanza di R tale che:

$$(i, A(i)) \in R \; \forall i \in dom(R)$$

4.2 Differenza tra Programma e Algoritmo

Si dice che un algoritmo risolve un problema [missing] Un programma differisce da un algoritmo in quanto: un programma può contenere diversi algoritmi, un programma è scritto in uno specifico linguaggio, il programma lavora su una struttura dati.
Algorithms + Data Structures = Programs

5 Peak Finding

Input: Un vettore A[0..n-1] di interi positivi.
Output: Un intero $0 \leq p \leq n$ tale che $A[p-1] \leq A[p] \geq A[p+1]$ dove $A[-1] = A[n] = -\infty$

Algorithm 1 Peak-Find-Left(A,n)

```
p ← 0
p ← 1
while k < n ∧ A[p] < A[k] do
  p ← k
  k ← k + 1
end while
return p
```

Nel caso migliore p=0 è un picco, nel caso peggiore il picco è l'elemento più a sinistra $p = n - 1$, si scorre in questo modo l'intero vettore effettuando $n - 1$ confronti.
Con lo stesso sforzo si trova il picco più alto, attraverso l'algoritmo di Peak finding-MAX.
Cosa garantisce quindi , di avere un picco nel vettore A[i..j] con $i \geq j$?

Ipotesi:
Se $A[i-1] \leq A[i] e A[j] \geq A[j+1]$, deve esserci allora un picco nel segmento A[i...j].

Teorema:
Siano i e j tali che :
$i \leq j$ e A[i..j] un vettore di n-interi.
Se $A[i-1] < A[i]$ e $A[j] > A[j+1]$ 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 allora , $A[i-1] \leq A[i] \geq A[i+1]$, p=i è **un picco.**
Se i e j sono diversi,
si sceglie una qualsia-
si posizione q , tale che
$i \leq q \leq j$

1. $A[q-1] \leq A[q] \geq A[q+1]$ **q è un picco**
2. $A[q-1] > A[q]$ **q non è un picco**
3. $A[q] < A[q+1]$ **q non è un picco**

Se q non è picco, siano :
$i_1 = i$ e $j_1 = q-1$ per $A[q-1] > A[q]$
$i_1 = q+1$ e $j_1 = j$ per $A[q-1] < A[q]$
si ha così:
$A[i_1 - 1] \leq A[i_1]$ e $A[j_1] \geq A[j_1 + 1]$, deve esserci allora un picco nel segmento $A[i_1...j_1]$.

Il nuovo segmento contiene meno elementi di quello precedente. Si ripete quindi la procedura descritta sopra, si sceglie un nuovo punto q_1, si verifica se $i_1 = j_1$ oppure se risultano diversi, in quel caso si dimezzerà ancora il segmento, fino a trovare un picco.

Tenere nota che q viene scelto **Arbitrariamente**, per facilitare i calcoli lo inizializzo a $q = n/2$, questo velocizza **esponenzialmente** l'esecuzione, ne consegue un algoritmo di tipo Divide et Impera.

Quanti controlli vanno eseguiti?
Si definisce una funzione tempo, la quale risulta implicita , perché T è definita in termini di se stessa, risulta una **relazione di ricorrenza**. Si usa il **Metodo dello Srotolamento** per esplicitarla.

Algorithm 2 Peak-DI(A,i,j)

Require: $i \leq j$
  $q \leftarrow [(i+j)/2]$
  if $A[q-1] \leq A[q] \geq A[q+1]$ **then**
    return p
  else if $A[q-1] > A[q] \vee A[q] < A[q+1]$ **then**
    if $A[q-1] > A[q]$ **then**
      return PeakFind-DI($A[i...q-1]$)
    else
      return PeakFind-DI($A[q+1...j]$)
    end if
  end if

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

Per esplicitazione otteniamo che

$$T(n) = 1 + \log_2 n$$

per $1 \leq k \leq \log_2 n$
Il tempo logaritmico è infinitesimalmente più piccolo del tempo lineare.

6 Problemi Insolubili

6.1 Problema dell'Halt

Halt(P,I) $\begin{cases} \text{true}, & \text{se P(I) termina} \\ \text{false}, & \text{altrimenti} \end{cases}$

Questa funzione termina solo se la funzione non termina. **Teorema dell'indecidibilità dell'Halt**

7 Problemi Intrattabili

8 Analisi Qualitativa

Avendo una formalizzazione di un problema viene fornita una nuova soluzione sotto forma di algoritmo in maniera **astratta**, per prima cosa si deve dimostrare che

l'algoritmo risolve il problema per tutte le istanze, poi esegue un calcolo della complessità. Se si completa la prima parte allora possediamo un **vero** algoritmo e si osserva se effettivamente termina, la seconda verrà osservata in seguito.

8.0.1 I Disastri Possibili

Durante la creazione di un algoritmo si possono presentare diverse situazioni indesiderate:

- Bugs: Letteralmente "insetti", errori di calcolo o logici causati da una cattiva programmazione da parte del progettatore.

8.1 Verificatore immaginario

Con la creazione di un algoritmo serve scoprire la presenza di un errore per poterlo individuare e negare la correttezza dell'algoritmo. La correttezza di un algoritmo su numero finito di iterazioni non implica la correttezza all'infinito. Viene quindi usata la **logica**.

8.2 Correttezza parziale e totale

Con qualunque ingresso *legale*:

- Se si raggiunge l'uscita ⇒ Correttezza **Parziale**.
- Se si raggiunge **sempre** l'uscita ⇒ Correttezza **Totale**.

Di norma viene prima controllata la correttezza parziale e solo in un secondo momento viene dimostrata la correttezza totale.

8.3 Specifica dell'algoritmo

- Pre-Condizione: Ipotesi di ingresso;
- Post-Condizione: Proprietà dell'uscita.

Queste condizioni servono a permette la validità di un immaginario *contratto* dove il client e l'algoritmo devono sottostare a delle specifiche regole per ottenere il corretto funzionamento. Tutto quello che è in pre-condizione va ipotizzato come **vero** e quindi non sarà controllato nell'algoritmo.

8.4 Ricorsione

Per facilitare la lettura e l'analisi dell'algoritmo, per convenzione, deve essere scritto senza salti e in maniera lineare.
Cos'è la ricorsione?

```
Una funzione è ricorsiva
se nella sua definizio-
ne utilizza direttamente o
indirettamente sé stessa
```

Durante un ricorsione, se abbiamo un caso in cui termina, abbiamo la garanzia che ad ogni step di ricorsione il caso diminuisce fino ad arrivare ad un **caso base**.

8.4.1 Le torri di Hanoi

Dati tre pioli su cui sono inseriti n dischi di diametro crescente, spostare la torre da un piolo sorgente ad un piolo destinazione, sfruttando un piolo d'appoggio muovendo un solo disco alla volta, senza mai sovrapporre un disco più grande ad uno più piccolo.

Algorithm 3 Hanoi(S,D,A)

Require: Abbiamo 3 pioli: Sorgente, Destinazione, Ausilio

if Sopra(sorgente) è vuota **then**
  Sposto l'unico disco di cui è fata sorgente su destinazione
else
  Hanoi(sopra(sorgente),ausilio,destinazione)
  Sposto l'unico disco di cui è fata sorgente su destinazione
  Hanoi(ausilio,sopra(destinazione),sorgente)
end if

8.5 Lo schema di induzione

Si divide in due parti:

- Il caso base: P(0)
- Il passo induttivo: $P(0) \Rightarrow$ P(n+1). L' ipotesi P(n) si chiama *ipotesi induttiva*

$$\frac{P(0) \qquad \forall m.P(m) \Rightarrow P(m+1)}{\forall n.P(n)}$$

8.5.1 Esempio di Ricorsione

Divisione ricorsiva:

$$a - b - b.... - b = r < b$$

diventa

$$(a - b) - b... - b = r < b$$

per cui

$$a - b = b\ q + r$$

avremo come resto $a - b$ con $a - b < a$ in quanto $b > 0$.

Algorithm 4 Div-Ric(a,b)

Require: $a \geq 0, b > 0$
Ensure:
  if $a < b$ **then**
  q, r $\leftarrow 0, a$
  else
    q', $r \leftarrow$ Div-Rec(a-b,b)
    $q \leftarrow q' + 1$
  end if
  return q,r

Quindi l'induzione completa con caso base: P(0)
e passo:
$\forall\, m[\forall n < m.P(n) \Rightarrow P(m)]$

$$\frac{[\forall m < n.P(m)] \Rightarrow P(n)}{\forall n.P(n)}$$

(Ricordarsi che nell'implicazione logica se l'antecedente [ipotesi] è falso, allora l'implicazione risulta vera). Spesso il caso base non viene specificato in quanto sottinteso nel calcolo.

8.6 Divisione Iterativa

A differenza della ricorsione è un percorso **lineare** dove ogni passo è della stessa forma, in quanto:

Iterazione: Ripetizione del Corpo.

Verifica funzionale controlla e verifica se un algoritmo si comporta nella maniera che noi ci aspettiamo e che restituisca il risultato desiderato.

L'algoritmo viene diviso in 3 parti:

- Si presuppone il valore/i desiderato/i;
- Si produce il calcolo funzionale dell'algoritmo, quale verrà ripetuto numero volte;
  - Viene protetto da una **guardia** che controlla una certa condizione ad ogni ciclo.
- Si raggiunge la condizione finale e termina l'algoritmo.

Le diverse variabili prendono determinati nomi in base alla loro funzione:

Parametri: Valori che non variano ad ogni funzione.

Variabili: Valori che mutano in base all'iterazione.

Accumulatore:

Algorithm 5 Div-It(a,b)

Require: $a > 0$, $b > 0$
Ensure: Return $q, r : a = b \cdot q + r \wedge 0 \leq r < b$
  $r \leftarrow a$
  $q \leftarrow 0$
  while $r \geq b$ **do**
    $r \leftarrow r - b$
    $q \leftarrow q + 1$
  end while
  // Inv: $a = bq + r \wedge 0 \leq r$
  return q, r

Invariante: Un asserto che pur cambiando i valori delle variabili, resta sempre vero.

8.7 Accumulatori ed Invarianti

Un accumulatore viene sempre inizializzato (pre entrare nel ciclo deve avere un valore). Nell'algoritmo vengono spesso usate delle *variabili di appoggio* ovvero delle variabili temporanee mirate ad evitare lo "sporcarsi" di variabili più importanti, questi dipendono dai valori genitori; alla fine del ciclo vengono sempre **re-inizializzate**.

8.8 Algoritmo di Euclide

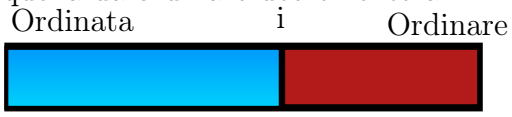
Algorithm 6 Algo di Euclide

$n = m \cdot q_0 + r_0$ // $0 < r_0 < m$
$m = r_0 \cdot q_1 + r_1$ // $0 < r_1 < r_0$
....
$r_{n-1} = r_n \cdot q_{n-1}$

Parte II
Algoritmi

1 Insertion Sort

Avendo una lista di elementi, impostiamo un valore i come indice, avrà lo scopo di separare la parte ordinata dal resto; "i" sarà la nostra **invariante di ciclo**. Il nucleo del processo è composto dal confronto dell'elemento A_i con quelli non ordinati; nel caso trovassimo un elemento minore del nostro A_i inizieremo a confrontarlo con gli elementi nella parte ordinata. Trovato il suo posto, avremo di nuovo una separazione tra parte ordinata e parte da ordinare, dove la parte ordinata incrementerà e quella da ordinare decrementerà.



Algorithm 7 Insertion-Sort(A)

```
for i ← 2 to length(A) do
    j ← i
    while j > i and A[j − 1] > A[j] do
        // Inv = ∀key ∈ A[j + 1..j].A[j] ≤ key
        scambia A[j − 1] con A[j] j ← j − 1
    end while
end for
return A
```

Quanto tempo richiede questo algoritmo per terminare?

$$t_i = \text{esecuzioni del while} \begin{cases} 1 & \text{nel caso migliore} \\ i & \text{nel caso peggiore} \end{cases}$$

$$T_{ins} = (c_1+c_2)n - c_2 + c_3 \sum_{i=2}^{n} i + (c_4+c_5) \sum_{i=2}^{n} (i-1)$$

Per amor di noi stessi la semplifichiamo:

$$\frac{c_3+c_4+c_5}{2} n^2 + (c_1+c_2+\frac{c_3-c_4-c_5}{2})n - (c_2+c_3)$$

ci permettiamo di mettere: $\frac{c_3+c_4+c_5}{2} = a$, $c_1+c_2+\frac{c_3-c_4-c_5}{2} = b$ $c_2+c_3 = c$ quindi:

$$an^2 + bn + c$$

da qui possiamo osservare che **La complessità temporale dell'Insertion Sort è *Quadratica*.**

2 Selection Sort

Algorithm 8 Select-Sort(A)

```
for i ← 1 to length (A) − 1 do
    k ← ifor j← i + 1 to length (A) do
        if A[k] > A[j] then
            k ← j
        end if
    end for
    scambia A[i] con A[k]
end for
return A
```

Complessità?

$$T_{set}(n) = \frac{1}{2}n^2 + \frac{7}{2}n - 3$$

(Calcolarsi per esercizio la tabella da soli)
Avrà complessità quadratica