Introduzione a problemi e algoritmi

February 22, 2017

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 regole 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 ← a mod b
3: while r ≠ 0 do
4:     a ← b
5:     b ← r
6:     r ← a mod b
7: end while
8: return b
```

Simulazione dell'algoritmo con $a = 20, b = 16$.

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 $n - 1 = 1048575$.

Correttezza e terminazione

February 22, 2017

Obiettivi: introdurre l'analisi qualitativa degli algoritmi, basata su tecniche di verifica come induzione e invarianti
Argomenti: i bug, correttezza (totale e parziale), verifica, pre- e postcondizioni, induzione ("semplice" e completa), verifica di algoritmi ricorsivi con induzione, invarianti di ciclo, verifica di algoritmi con invarianti di ciclo, terminazione.

1 Bug, correttezza, verifica, pre- e postcondizioni

I bug, termine inventato da Edison, sono inevitabili: "It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise — this thing gives out and [it is] then that "Bugs" — as such little faults and difficulties are called — show themselves and months of intense watching, study and labor are requisite before commercial success or failure is certainly reached." (T. Edison, da Wikipedia)

Le loro conseguenze possono essere devastanti:
1962: la sonda Mariner 1 si schianta su Venere per un errore nel software di controllo del volo
1981: una TV nel Quebec decreta la vittoria alle elezioni di un partito sconosciuto; usavano un software difettoso per la rilevazione dei risultati
1983: un sistema computerizzato sovietico rileva un attacco nucleare con cinque missili balistici inesistenti
1985: alcuni pazienti vengono irradiati con dosi eccessive di raggi X dal sistema Therac-25 a controllo software
1993: il processore Pentium, che incorpora il coprocessore aritmetico, sbaglia le divisioni in virgola mobile
1996: l'Ariane 5 esce dalla sua rotta e viene distrutto in volo a causa di un errore di conversione dei dati da 16 a 32 bit
1999: usando un nuovo sistema, le poste inglesi recapitano mezzo milione di nuovi passaporti ad indirizzi sbagliati o inesistenti
1999: Y2K è il celeberrimo millenium-bug
(continua...)

Bisogna **verificare la correttezza degli algoritmi e i programmi** che li implementano.

Un algoritmo (un programma) è **corretto** se per ogni input **fornisce l'output corretto** (correttezza totale).

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

Un "verificatore ideale", dato un problema P a un algoritmo A, dovrebbe essere in grado di rispondere:
"**SI**, A è corretto per P" oppure "**NO**, A non è corretto per P e gli errori sono...".

Tale verificatore non esiste ma ci sono tanti progetti di ricerca che puntano a sviluppare strumenti per aiutare la verifica di algoritmi e programmi.

Verificare un algoritmo (un programma) vuole dire controllare se esso soddisfa le specifiche del problema computazionale che deve risolvere. Le specifiche sono descritte tramite le **precondizioni** e le **postcondizioni**.

Per esempio, per il problema della divisione intera:
Precondizioni: $a \geq 0, b > 0$ numeri interi;
Postcondizioni: il risultato è un intero q tale che $a = b \cdot q + r$ con $0 \leq r < b$.

2 Induzione "semplice" e completa

Schema dell'**induzione "semplice"**:
Sia $P(n)$ una proprietà che dipende da una variabile intera $n \in \mathbb{N}$.
Se $P(n)$ vale per $n = 0$ e per ogni $n \in \mathbb{N}$ vale $P(n) \implies P(n+1)$ allora $P(n)$ vale per ogni $n \in \mathbb{N}$.

Cioè

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

dove $P(0)$ è il **caso base** e $(\forall n \in \mathbb{N}.P(n) \implies P(n+1))$ è il **passo induttivo**. Inoltre, $P(n)$ viene chiamata l'**ipotesi induttiva**.

In parole: una proprietà $P(n)$ che vale per $n = 0$ (caso base) e se vale per n allora vale per $n + 1$ (passo induttivo) vale per ogni $n \geq 0$.

Il caso base può essere anche $P(k)$:

$$(P(k) \wedge (\forall n \geq k.P(n) \implies P(n+1))) \implies \forall n \geq k.P(n)$$

quindi la proprietà vale da k in poi.

Esempio: dimostriamo con induzione che

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

Caso base: la proprietà con $n = 1$ vale

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

Passo induttivo: supposta che la proprietà in (1) è vera per n (questa è l'ipotesi induttiva), verifichiamo per $n + 1$

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

dove abbiamo sostituito $(\sum_{k=1}^{n}(2k-1))$ con n^2 grazie all'ipotesi induttiva. Visto che la proprietà in (1) vale per $n = 1$ (caso base) e se vale per n allora vale per $n+1$ (passo induttivo), la proprietà vale per qualunque $n \geq 1$.

Schema dell'**induzione completa**:
Sia $P(n)$ una proprietà che dipende da una variabile intera $n \in \mathbb{N}$.
Se $P(n)$ vale per $n = 0, 1, ..., k$ e per ogni $n > k$ vale $(P(0) \wedge P(1) \wedge ... \wedge P(n)) \implies P(n+1)$ allora $P(n)$ vale per ogni $n \in \mathbb{N}$.

Cioè

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

dove $P(0) \wedge P(1) \wedge ... \wedge P(k)$ è il **caso base** e $(\forall n > k.(P(0) \wedge P(1) \wedge ... \wedge P(n)) \implies P(n+1))$ è il **passo induttivo**. Inoltre, $P(0) \wedge P(1) \wedge ... \wedge P(n)$ viene chiamata l'**ipotesi induttiva**.

In parole: una proprietà $P(n)$ che vale per $n = 0, 1, ..., k$ (caso base) e se vale per $0, 1, ..., n$ allora vale per $n + 1$ (passo induttivo) vale per ogni $n \geq 0$.

Dimostriamo la seguente teorema con induzione completa.

Teorema: Per ogni numero naturale $n \geq 2$ esiste una sequenza $p_1, p_2, ..., p_k$ tale che $\prod_{i=1}^{k} p_i = n$ e $\forall i, 1 \leq i \leq k.p_i \in \mathbb{P}$, cioè ogni numero naturale $n \geq 2$ è prodotto di numero primi.
Dimostrazione: Procediamo con induzione su n.
Caso base. L'asserto è vero per $n = 2$ perché 2 è un numero primo.
Passo induttivo. Dobbiamo dimostrare che se l'asserto è vero per ogni $n' \in \{2, 3, ..., n\}$ 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 due numeri più piccoli n_1, n_2, cioè $n+1 = n_1 \cdot n_2$ con $1 < n_1 < n$ e $1 < n_2 < n$
- per ipotesi induttiva, n_1 e n_2 sono esprimibili come prodotto di numeri primi:

$$n_1 = \prod_{i=1}^{s} q_i, \quad \forall i, 1 \leq i \leq s. q_i \in \mathbb{P}, \quad n_2 = \prod_{i=1}^{t} r_i, \quad \forall i, 1 \leq i \leq t. r_i \in \mathbb{P}$$

  e sostituendo

$$n+1 = n_1 \cdot n_2 = \prod_{i=1}^{s} q_i \cdot \prod_{i=1}^{t} r_i$$

  che è un prodotto di numeri primi e quindi il passo induttivo vale.

Per il principio dell'induzione completa, l'asserto è vera per ogni $n \geq 2$. □

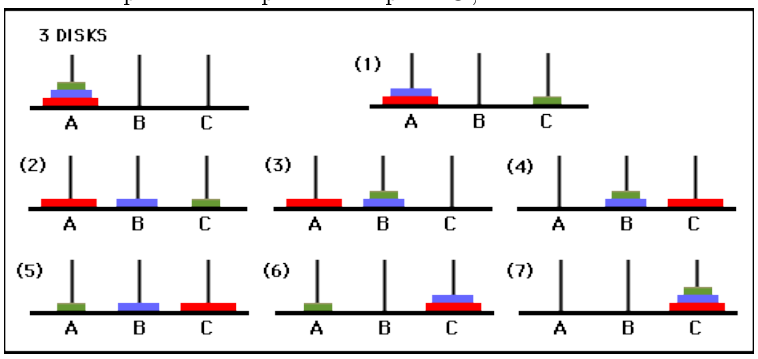
3 Dimostrazione di correttezza di algoritmi ricorsivi con induzione

Il problema delle torre di Hanoi. Dati tre pioli

- il piolo *sorgente* con sopra n dischi di diametro crescente,
- il piolo *d'appoggio* senza dischi,
- e il piolo *destinazione* senza dischi,

il compito è spostare la torre dal piolo sorgente al piolo destinazione, sfruttando il piolo d'appoggio, muovendo un disco alla volta, senza mai sovrapporre un disco più grande ad uno più piccolo.

Per esempio, con tre dischi da spostare dal piolo A al piolo C, le mosse sono $2^2 - 1$:



Sviluppiamo un algoritmo ricorsivo per risolvere il problema in presenza di n dischi.

Caso base. Quando il numero di dischi da spostare è 1, il problema è triviale.

Ricorsione. Supponendo che siamo in grado di risolvere il problema correttamente con $n-1$ dischi, si può risolvere il problema con n dischi in tre passaggi:
1. sposta $n-1$ dischi dal piolo sorgente al piolo d'appoggio;
2. sposta un disco dal piolo sorgente al piolo destinazione;
3. sposta $n-1$ dischi dal piolo d'appoggio al piolo destinazione.

In pseudo-codice:

```
1: MoveTower(n, A, B, C)      ▷ n dischi da spostare dal piolo A al piolo C utilizzando
                                il piolo B come appoggio;
                                Precondizione: A non è vuota e la base di A ha
                                un diametro più piccolo del disco più in alto sia di
                                B che di C;
                                Postcondizione: n dischi sono spostati da dal piolo
                                A al piolo C;
2: if n = 1 then
3:     move 1 disk from A to C
4: else
5:     MoveTower(n − 1, A, C, B)
6:     move 1 disk from A to C
7:     MoveTower(n − 1, B, A, C)
8: end if
```

Teorema: L'algoritmo precedente è corretto.
Dimostrazione: Procediamo con induzione su n.
Caso base. Con $n = 1$ è triviale che l'algoritmo sia corretto.
Passo induttivo. Dobbiamo dimostrare che se l'algoritmo è corretto con n dischi (ipotesi induttiva) allora è corretto con $n + 1$ dischi. Con $n + 1$ dischi:

- all'inizio i numeri di dischi sui pioli sono: $n + 1, 0, 0$
- MoveTower$(n - 1, A, C, B)$ sposta, grazie all'ipotesi induttiva, correttamente n dischi dal piolo A al piolo B; dopodiché abbiamo $1, n, 0$ dischi sui pioli
- dopo "move 1 disk from A to C" la situazione è $0, n, 1$
- MoveTower$(n - 1, B, A, C)$ sposta, grazie all'ipotesi induttiva, correttamente n dischi dal piolo B al piolo C; dopodiché abbiamo $0, 0, n + 1$ dischi sui pioli.

Quindi se l'algoritmo è corretto con n dischi allora è corretto con $n + 1$ dischi.
Per il principio di induzione, l'algoritmo è corretto per ogni $n \geq 1$. □

Il problema della divisione intera.
Precondizioni: $a \geq 0, b > 0$ numeri interi;
Postcondizioni: i risultati sono due interi q, r tali che $a = b \cdot q + r$ con $0 \leq r < b$ (dove q è il quoziente e r è il resto).

Secondo le postcondizioni dobbiamo avere $a = b \cdot q + r$ con $0 \leq r < b$, cioè

$$a - q \cdot b = a - \underbrace{b - b - ... - b}_{q} = r < b$$

Se $a < b$ allora la soluzione è $q = 0$ con $r = a$. Se $a \geq b$ allora

$$(a - b) - \underbrace{b - b - ... - b}_{q'} = r < b$$

e quindi q' e r sono quoziente e resto di $a - b$ diviso b e $q = q' + 1$.

Visto che $a - b < a$, il precedente ragionamento indica un procedimento ricorsivo per risolvere il problema:

```
1: Div-Rec(a, b)      ▷ Pre: a ≥ 0, b > 0
                        Post: ritorna q, r tali che a = b q + r ∧ 0 ≤ r < b
2: if a < b then
3:     q, r ← 0, a
4: else
5:     q′, r ← Div-Rec(a − b, b)
```

6: $q \leftarrow q' + 1$
7: **end if**
8: **return** q, r

Teorema: Il precedente algoritmo è corretto.
Dimostrazione: Fissato $b > 0$, procediamo con induzione completa su a.
Casi base. Se $a < b$, cioè per ogni $a \in \{0, 1, ..., b-1\}$ l'algoritmo restituisce $q = 0, r = a$ che è chiaramente corretto.
Passo induttivo. Dobbiamo dimostrare che, in caso di $a \geq b$, se l'algoritmo è corretto per ogni $a' < a$ allora l'algoritmo è corretto per a:

- se per $a' = a - b, b$ l'algoritmo restituisce q', r' allora per a, b restituisce $q = q' + 1, r = r'$
- secondo l'ipotesi induttiva la risposta per a', b è corretta e quindi

  $$a' = a - b = b \cdot q' + r' \wedge 0 \leq r' < b$$

  da dove segue che

  $$a = b \cdot q' + b + r' = b(q' + 1) + r' \wedge 0 \leq r' < b$$

  e visto che $q = q' + 1, r = r'$ abbiamo

  $$a = b \cdot q + r \wedge 0 \leq r < b$$

  e quindi il passo induttivo è dimostrato.

Per il principio di induzione completa, l'algoritmo è corretto per qualunque $a \geq 0, b > 0$. □

4 Dimostrazione di correttezza di algoritmi iterativi con invarianti

L'**invariante di ciclo** è una proposizione sul valore delle variabili "intorno" ad un ciclo con le seguenti proprietà:

- *inizializzazione:* la proposizione vale immediatamente prima di entrare nel ciclo;
- *mantenimento:* se la proposizione vale prima di eseguire il corpo del ciclo, allora vale anche dopo aver eseguito il corpo del ciclo.

L'invariante di ciclo è utile se, al termine del ciclo, aiuta a dimostrare la correttezza dell'algoritmo.

Esempio. Algoritmo iterativo per calcolare l'ennesimo numero della seria di Fibonacci ($F_1 = F_2 = 1, F_n = F_{n-2} + F_{n-1}, \forall n \geq 2$).

1: FIB-ITE(n) ▷ Pre: $n \geq 1$
Post: ritorna F_n
2: $a, b, i \leftarrow 1, 1, 3$
3: **while** $i \leq n$ **do**
4: $c \leftarrow a + b$
5: $a \leftarrow b$
6: $b \leftarrow c$
7: $i \leftarrow i + 1$
8: **end while**
9: **return** b

Teorema: l'invariante, che vale ogni volta prima di eseguire la riga 3 e ogni volta dopo aver eseguito la riga 7, è

$$b = F_{i-1} \wedge a = F_{i-2}$$

L'invariante garantisce la correttezza dell'algoritmo.
Dimostrazione: È immediato che l'invariante vale prima di entrare nel ciclo: $a = 1, b = 1, i = 3$ e quindi $b = F_2 \wedge a = F_1$.
Il ciclo mantiene l'invariante. Assumiamo che prima di eseguire il corpo del ciclo abbiamo $b = F_{i-1} \wedge a = F_{i-2}$. Alla variabile c viene assegnato $a + b = F_{i-2} + F_{i-1} = F_i$, alla variabile a viene assegnate $b = F_{i-1}$ e alla variabile b il valore $c = F_i$. Visto che i viene incrementato, dopo l'esecuzione del corpo del ciclo abbiamo di nuovo $b = F_{i-1} \wedge a = F_{i-2}$.
All'uscita del ciclo si ha $i = n + 1$, e quindi all'uscita $b = F_{i-1} \wedge a = F_{i-2}$ implica

$$b = F_{(n+1)-1} = F_n$$

e dunque l'algoritmo è corretto. □

5 Terminazione

Ordinamento (algoritmi quadratici)

March 3, 2017

Obiettivi: attraverso lo studio degli algoritmi di ordinamento quadratici, applicare la nozione di invariante e introdurre l'analisi di complessità.
Argomenti: problema dell'ordinamento, insertion-sort e selection-sort con analisi di correttezza e complessità.

1 Problema dell'ordinamento (sorting)

La ricerca in un vettore di n elementi richiede n confronti nel caso peggiore

Se il vettore è ordinato, si può applicare la ricerca binaria (ricerca dicotomica) che richiede al più $\log_2 n$ confronti:

```
BinSearch-Ric(x, A, i, j)
    ▷ Pre: A[i..j] ordinato
    ▷ Post: true se x ∈ A[i..j]
if i > j then        ▷ A[i..j] = ∅
    return false
else
    m ← ⌊(i + j)/2⌋
    if x = A[m] then
        return true
    else
        if x < A[m] then
            return BinSearch-Ric(x, A, i, m − 1)
        else      ▷ A[m] < x
            return BinSearch-Ric(x, A, m + 1, j)
        end if
    end if
end if
```

Per ordinare un vettore, si potrebbe pensare di generare tutte le permutazioni e scegliere quella nella quale gli elementi sono ordinati:

```
Sorted(A[1..n])
for i ← 2 to n  do
    if A[i − 1] > A[i] then
        return false
    end if
end for
return true
```

```
TRIVIAL-SORT(A)
for all A' permutazione di A do
    if SORTED(A') then
        return A'
    end if
end for
```

Ma il numero di permutazioni è $n!$ che cresce più velocemente di 2^n. Non è praticabile.

2 Insertion-sort

L'idea generale: data un vettore con la parte sinistra, $A[1..i-1]$, ordinata, è facile inserirci l'elemento $A[i]$ in modo tale che il sottovettore $A[1..i]$ risulti ordinata (aumentando così la parte ordinata). Punto di partenza: $i = 2$ (così la parte a sinistra di $A[i]$ contiene un singolo elemento e quindi è ordinata).

```
0: INSERTION-SORT(A[1..n])
1: for i ← 2 to n do
2:     j ← i
3:     while j > 1 and A[j − 1] > A[j] do
4:         scambia A[j − 1] con A[j]
5:         j ← j − 1
6:     end while
7: end for
```

2.1 Dimostrazione della correttezza con invarianti

Invariante del ciclo esterno:

$$\text{il sottovettore } A[1..i-1] \text{ è ordinato.}$$

Inizializzazione: con $i = 2$ è vero (fa riferimento ad un vettore di un singolo elemento).
Mantenimento: assumendo che il ciclo interno inserisci al posto giusto $A[i]$ (la dimostreremo di seguito) l'invariante viene mantenuta.
All'uscita: con $i = n + 1$ l'invariante implica che il vettore è ordinato.

Invariante del ciclo interno:

$$\text{il vettore composto da } A[1..j-1] \text{ e } A[j+1..i] \text{ è ordinato } \wedge\ \forall k, j+1 \leq k \leq i . A[j] < A[k]$$

cioè, per quanto riguarda il segmento $A[1..i]$, togliendo $A[j]$ si ottiene un vettore ordinato e gli elementi dopo $A[j]$ sono maggiori di $A[j]$.

Inizializzazione: grazie all'invariante del ciclo esterno l'invariante vale.
Mantenimento: come ipotesi induttiva assumiamo che l'invariante vale prima di eseguire il corpo del ciclo; ci sono due possibilità:

1. se $A[j-1] \leq A[j] \vee j = 1$ allora si esce dal ciclo e all'uscita, grazie al fatto che $A[j-1] \leq A[j] \leq A[j+1]$ (dove la parte destra c'è solo so $j \neq i$ e la parte sinistra solo se $j \neq 1$) e all'ipotesi induttiva, tutto il sotto vettore $A[1..i]$ è ordinato,
2. se $A[j-1] < A[j]$ allora si fa lo scambio fra $A[j-1]$ e $A[j]$ e l'invariante rimane valido

All'uscita: discusso al punto 1. sopra.

Quindi l'algoritmo INSERTION-SORT è corretto.

2.2 Tempo di esecuzione del INSERTION-SORT

Calcoliamo il tempo di esecuzione assumendo che eseguire la riga $i, 1 \leq i \leq 5$, una volta richiede c_i unità di tempo.

La riga 1 viene eseguita con $i = 2, 3, ..., n, n+1$, cioè n volte (viene eseguita con $i = n+1$ perché bisogna accorgersi di dover uscire dal ciclo).
La riga 2 viene eseguita con $i = 2, 3, ..., n$, cioè $n-1$ volte.
Dato un valore per i, denotiamo con t_i il numero di volte che la riga 3 viene eseguita. Nel caso migliore $t_i = 1$ (non servono scambi per inserire $A[i]$ al posto giusto) e nel caso peggiore $t_i = i$ (bisogna portare $A[i]$ all'inizio del vettore e quindi la condizione del while viene valutata con $j = i, i-1, ..., 2, 1$). In totale la riga 3 viene eseguita $\sum_{i=2}^{n} t_i$ volte.
La riga 4 viene eseguita $t_i - 1$ volta dato un valore per i (una volta in meno rispetto alla riga 3). In totale la riga 4 viene eseguita $\sum_{i=2}^{n}(t_i - 1)$ volte.
La riga 5 viene eseguita $t_i - 1$ volta dato un valore per i (una volta in meno rispetto alla riga 3). In totale la riga 5 viene eseguita $\sum_{i=2}^{n}(t_i - 1)$ volte.

Considerando il caso peggiore (cp) il tempo di esecuzione è

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

$$(c_1 + c_2)n - c_2 + c_3 \frac{2+n}{2}(n-1) + (c_4 + c_5)\frac{1+n-1}{2}(n-1)$$

Segue che $T_{cp}(n)$ è un polinomio di secondo grado in n, cioè può essere espresso come

$$T_{cp}(n) = an^2 + bn + c$$

Di conseguenza, per valori grandi di n,

$$T_{cp}(n) \approx an^2$$

e quindi il **tempo di esecuzione nel caso peggiore è proporzionale al quadrato del numero di elementi** del vettore. Nel caso peggiore l'algoritmo è quadratico.

Considerando il caso migliore (cm) il tempo di esecuzione è

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

$$(c_1 + c_2)n - c_2 + c_3(n-1)$$

Segue che $T_{cp}(n)$ è un polinomio di primo grado in n, cioè può essere espresso come

$$T_{cp}(n) = an + b$$

Di conseguenza, per valori grandi di n,

$$T_{cp}(n) \approx an$$

e quindi il **tempo di esecuzione nel caso migliore è proporzionale al numero di elementi** del vettore. Nel caso migliore l'algoritmo è lineare.

3 Selection-sort

L'idea generale: data un vettore in cui la parte sinistra, $A[1..i-1]$, è ordinata e contiene gli $i-1$ numeri più piccoli del vettore, si cerca il minimo della parte $A[i..n]$ e si mette nella posizione i (aumentando così la parte ordinata). Punto di partenza: $i = 1$ (così la parte a sinistra di $A[i]$ è un vettore "vuoto").

```
0: SELECTION-SORT(A[1..n])
1: for i ← 1 to n − 1 do
2:     k ← i
3:     for j ← i + 1 to n do
4:         if A[k] > A[j] then
5:             k ← j
6:         end if
7:     end for
8:     scambia A[i] con A[k]
9: end for
```

3.1 Dimostrazione della correttezza con invarianti

Invariante del ciclo esterno:

$$\text{il sottovettore } A[1..i-1] \text{ è ordinato } \wedge\ \forall k, l, 1 \leq k \leq i-1, i \leq l \leq n. A[k] \leq A[l]$$

Inizializzazione: con $i = 1$ la proposizione è "vuota" e quindi vale.
Mantenimento: assumendo che il ciclo interno selezioni il minimo del $A[i..n]$ correttamente (la discuteremo di seguito) l'invariante viene mantenuta.
All'uscita: con $n = 1$ l'invariante implica

$$\text{il sottovettore } A[1..n-1] \text{ è ordinato } \wedge\ A[n-1] \leq A[n]$$

e quindi il vettore è ordinato.

Invariante del ciclo interno:

$$A[k] \text{ è il minimo in } A[i..j-1]$$

Inizializzazione: con $k = i$ e $j = i + 1$ è triviale.
Mantenimento: come ipotesi induttiva assumiamo che l'invariante vale prima di eseguire il ciclo; il corpo del ciclo aggiorna la posizione del massimo se $A[k] > A[j]$ e poi in ogni caso incrementa j; quindi l'invariante viene mantenuta.
All'uscita: con $j = n + 1$ l'invariante implica che $A[k]$ è il minimo in $A[i..n]$ e quindi il ciclo interno funziona correttamente.

Quindi l'algoritmo SELECTION-SORT è corretto.

3.2 Tempo di esecuzione del SELECTION-SORT

Deriviamo il tempo di esecuzione calcolando quante volte vengono eseguite le righe 1,2,3,4,5 e 8. Associamo con ognuna di queste righe un tempo di esecuzione uguale a 1 unità di tempo, cioè $c_1 = c_2 = c_3 = c_4 = c_5 = c_8 = 1$. Questa scelta non cambia in che modo il tempo di esecuzione dipende dal numero di elementi del vettore.

Nel caso peggiore l'indice del minimo in $A[i..n]$ viene aggiornato ogni volta (cioè la condizione della riga 4 risulta sempre "true"):

$$T_{cp}(n) = \underbrace{n}_{\text{riga 1}} + \underbrace{n-1}_{\text{riga 2}} + \underbrace{\sum_{i=1}^{n-1}(n-(i+1)+1+1)}_{\text{riga 3}} + \underbrace{\sum_{i=1}^{n-1}(n-(i+1)+1)}_{\text{riga 4}} + \underbrace{\sum_{i=1}^{n-1}(n-(i+1)+1)}_{\text{riga 5}} + \underbrace{n-1}_{\text{riga 8}} =$$

$$\underbrace{n}_{\text{riga 1}} + \underbrace{n-1}_{\text{riga 2}} + \underbrace{\sum_{i=1}^{n-1}(n-i+1)}_{\text{riga 3}} + \underbrace{\sum_{i=1}^{n-1}(n-i)}_{\text{riga 4}} + \underbrace{\sum_{i=1}^{n-1}(n-i)}_{\text{riga 5}} + \underbrace{n-1}_{\text{riga 8}} =$$

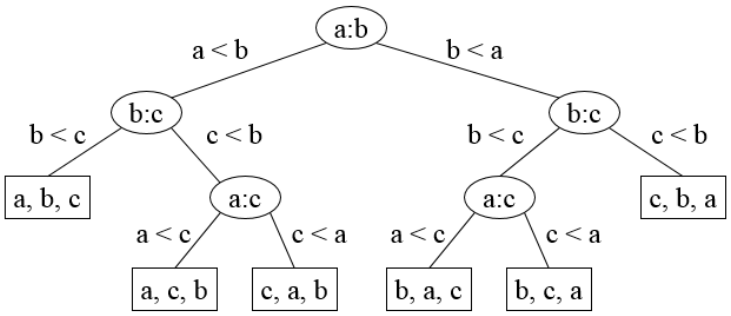
$$\underbrace{n}_{\text{riga 1}} + \underbrace{n-1}_{\text{riga 2}} + \underbrace{\sum_{i=1}^{n-1}(i+1)}_{\text{riga 3}} + \underbrace{\sum_{i=1}^{n-1} i}_{\text{riga 4}} + \underbrace{\sum_{i=1}^{n-1} i}_{\text{riga 5}} + \underbrace{n-1}_{\text{riga 8}}$$

La precedente è un polinomio di secondo grado in n. Per grandi valori di n, il tempo di esecuzione è proporzionale al quadrato del numero di elementi. Quindi **nel caso peggiore l'algoritmo è quadratico**.

Nel caso migliore il termine associato con la riga 5 è 0 (non viene mai aggiornato la posizione del minimo). Nonostante ciò, il tempo di esecuzione, $T_{cm}(n)$, è un polinomio di secondo grado in n. Quindi anche **nel caso migliore l'algoritmo è quadratico**.

4 Minimo numero di confronti per ordinare un vettore

I confronti che si devono fare per ordinare un vettore si possono rappresentare con alberi binari di decisione in cui i nodi interni rappresentano i confronti e le foglie i possibili ordini finali. Il seguente albero rappresenta i confronti necessari per ordinare un vettore di tre elementi.



Tale albero, per un vettore di n elementi, ha $n!$ foglie (tutte le possibili permutazioni devono esserci).

Con $n!$ foglie, il ramo più lungo contiene almeno $\lceil \log_2 n! \rceil$ nodi interni. (Nel caso di $n = 3$, abbiamo $\lceil \log_2 3! \rceil = 3$.)

Ciò significa che un algoritmo di ordinamento per confronti deve fare nel caso peggiore almeno $\lceil \log_2 n! \rceil$ confronti.

Secondo la formula di Stirling, per grandi valori di n, si può approssimare $n!$ con

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Dopo qualche algebra segue che, per grandi valori di n, **il numero di confronti che un algoritmo di ordinamento per confronti deve fare nel caso peggiore è proporzionale a**

$$\mathbf{n \log_2 n}$$