Algoritmi e Strutture Dati

Un'introduzione informale agli algoritmi

Basato su materiale di C. Demetrescu, I. Finocchi, G.F. Italiano

Definizione informale di algoritmo

Insieme di istruzioni tali da poter essere eseguite meccanicamente e da produrre un determinato risultato

algoritmo `preparaCaffè`

1. Svita la caffettiera.
2. Riempi d'acqua il serbatoio della caffettiera.
3. Inserisci il filtro.
4. Riempi il filtro con la polvere di caffè.
5. Avvita la parte superiore della caffettiera.
6. Metti la caffettiera, così predisposta, su un fornello acceso.
7. Spegni il fornello quando il caffè è pronto.
8. Versa il caffè nella tazzina.

L'algoritmo risolve un problema

- IN INFORMATICA: problemi di elaborazione di informazioni

Problemi, Algoritmi, Programmi

- Problema
  - specifica una funzione: dominio di input, dominio di output (risultati possibili), e mapping fra istanze in input e risultati desiderati
  - vari tipi di problemi (decisione, ricerca, ottimizzazione)
- Algoritmo
  - procedimento per il calcolo della funzione, su una qualunque istanza di input, codificato in termini di istruzioni (operazioni elementari) riconosciute dall'esecutore
  - Per un problema possono esistere, uno o più algoritmi, o nessuno ("non risolvibilità" o "incomputabilità")
- Programma
  - Traduzione di un algoritmo in un particolare linguaggio di programmazione
  - L'esecuzione di un algoritmo su un particolare calcolatore elettronico ne richiede la traduzione nel linguaggio macchina di questo
  - E' possibile usare linguaggi di programmazione di alto livello

Un problema non risolvibile

Problema delle Corrispondenze:

Dati due insiemi di stringhe S1 e S2, trovare una stringa x tale che x sia:

- la concatenazione di stringhe in S1 e
- la concatenazione di stringhe in S2

Esempio:

Dati $S1 = \{01, 00, 110\}$ **e** $S2 = \{001, 1\}$

una soluzione è: $0011001 = \begin{cases} 00 \cdot 110 \cdot 01 \\ 001 \cdot 1 \cdot 001 \end{cases}$

Qualità di un algoritmo

Correttezza:

l'algoritmo produce il risultato desiderato

Efficienza:

l'algoritmo fa un uso parsimonioso delle risorse di calcolo

Algoritmi poco efficienti possono essere inutilizzabili nella pratica (richiedono troppe risorse)

Complessità (costo) di un algoritmo

Quantità di risorse usate per l'esecuzione

- costo temporale: tempo (numero di "operazioni")
- costo spaziale: quantità di memoria aggiuntiva (numero di celle elementari) a quella usata per rappresentare l'input

Esempio

- Problema: indovinare un numero compreso tra 1 e 100
  - *Algoritmo 1: tentativi casuali*
  - *Algoritmo 2: procedere di 10 in 10 e poi di 1 in 1*
  - *Algoritmo 3: ricerca binaria*
- Il diverso numero medio di tentativi ci suggerisce di adottare l'algoritmo 3

Complessità di un problema

- La progettazione di algoritmi sempre più efficienti trova il limite nella difficoltà "intrinseca del problema"

  Es: non esiste un metodo migliore dell'*Algoritmo 3* per indovinare in modo *"sistematico"* un numero compreso tra *1* e *n* (servono circa $log_2(n)$ tentativi)

- Esistono classi di problemi "difficili" ("intrattabili") le cui implementazioni sono necessariamente onerose
  - Verificare se un numero è primo è "difficile" (intrattabile)
  - Scomporre un numero in fattori primi è "difficile" (intrattabile)
  - Trovare il minimo comune multiplo di due numeri è facile (trattabile)

Analisi di complessità

- I costi dipendono dal modello di calcolo (esecutore)
- Per una valutazione "ragionevole" di tale costo bisogna determinare il legame esistente tra la **"dimensione dell'input"** e le risorse impegnate
- La valutazione può essere fatta per i casi migliore, peggiore e medio

Modello di macchina

- Un specifico calcolatore elettronico reale
  - *Il costo dipende dalla macchina scelta*
  - *Non si può ottenere una legge che ci dica come varia tale costo al variare della macchina*
- Modelli astratti di computazione
  - *macchine di Turing*
  - *macchine a registri*
  - *uso di un linguaggio ad alto livello*

Analisi teorica della complessità

- L'analisi teorica è più completa e generale di quella sperimentale (tempi misurati su un calcolatore):
  - vale su tutte le possibili istanze di dati di input, e non dipende dal particolare linguaggio/calcolatore usato
- Fondamentale per la progettazione di SW
  - Permette di predire le prestazioni di un programma software, prima ancora di scriverne le prime linee di codice
  - Individuare le funzionalità più critiche
  - Aiuta a scegliere la soluzione migliore per un dato problema

Pseudocodice

- Per mantenere il massimo grado di generalità, descriveremo gli algoritmi in pseudocodice:
  - ricorda linguaggi di programmazione reali come C, C++ o Java
  - può contenere frasi in italiano

La traduzione in un particolare linguaggio di programmazione può essere fatta in modo quasi meccanico

Un esempio: numeri di Fibonacci

Leonardo da Pisa (anche noto come Fibonacci) si interessò di molte cose, tra cui il seguente problema di dinamica delle popolazioni:

Quanto velocemente si espanderebbe una popolazione di conigli sotto appropriate condizioni?

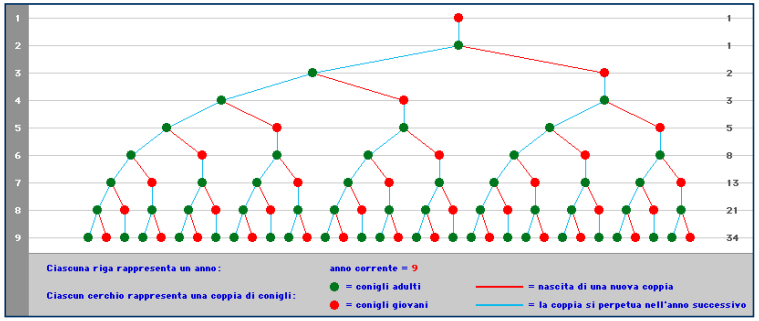
In particolare, partendo da una coppia di conigli in un'isola deserta, quante coppie si avrebbero nell'anno n?

Le regole di riproduzione

- Una coppia di conigli genera due coniglietti ogni anno
- I conigli cominciano a riprodursi soltanto al secondo anno dopo la loro nascita
- I conigli sono immortali

L'albero dei conigli

La riproduzione dei conigli può essere descritta in un albero come segue:



La regola di espansione

- Nell'anno *n*, ci sono tutte le coppie dell'anno precedente, e una nuova coppia di conigli per ogni coppia presente due anni prima
- Indicando con F_n il numero di coppie dell'anno *n*, abbiamo la seguente relazione di ricorrenza:

$$\mathbf{F_n} = \begin{cases} \mathbf{F_{n-1}} + \mathbf{F_{n-2}} & \text{se } n \geq 3 \\ \mathbf{1} & \text{se } n = 1,2 \end{cases}$$

Problema: Calcolare F_n

Un approccio numerico

- Possiamo usare una funzione matematica che calcoli direttamente i numeri di Fibonacci.
- Si può dimostrare che:

$$F_n = \frac{1}{\sqrt{5}}\left(\phi^n - \hat{\phi}^n\right) \qquad \begin{aligned} \phi &= \tfrac{1+\sqrt{5}}{2} \approx +1.618 \\ \hat{\phi} &= \tfrac{1-\sqrt{5}}{2} \approx -0.618 \end{aligned}$$

- Un possibile algoritmo:

algoritmo `fibonacci1`$(intero\ n) \rightarrow intero$
return $\frac{1}{\sqrt{5}}\left(\phi^n - \hat{\phi}^n\right)$

Correttezza?

- Qual è l'accuratezza su Φ e Φ per ottenere un risultato corretto?
- Ad esempio, con 3 cifre decimali:

$$\phi \approx 1.618 \text{ e } \hat{\phi} \approx -0.618$$

n	fibonacci1(n)	arrotondamento	F_n
3	1.99992	2	2
16	986.698	987	987
18	2583.1	2583	2584

Algoritmo fibonacci2

Applichiamo la definizione ricorsiva (operando solo su numeri interi)

algoritmo fibonacci2*(intero n)* → *intero*
  if (n≤2) **then return** 1
  else return fibonacci2(n-1) + fibonacci2(n-2)

Tempo di esecuzione: *numero di linee di codice* eseguite

- Se n≤2: una sola linea di codice
- Se n=3: 4 linee di codice (2 per fibonacci2(3) + 1 per fibonacci2(2) + 1 per fibonacci2(1))

Misura rozza, ma indipendente dalla piattaforma utilizzata …

Relazione di ricorrenza

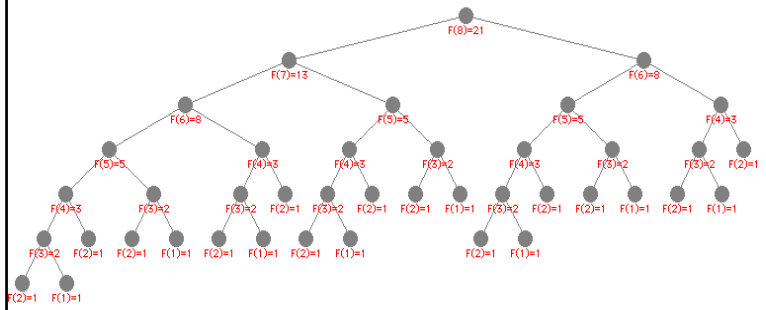
Il tempo richiesto da un algoritmo ricorsivo è pari a:
-il tempo speso all'interno della chiamata
-più il tempo speso nelle chiamate ricorsive

$$T(n) = 2 + T(n-1) + T(n-2)$$

algoritmo fibonacci2*(intero n)* → *intero*
  if (n≤2) **then return** 1
  else return fibonacci2(n-1) + fibonacci2(n-2)

Albero della ricorsione

- I nodi corrispondono alle chiamate ricorsive
- I figli di un nodo corrispondono alle sottochiamate



Calcolare T(n)

- Etichettando i nodi dell'albero con il numero di linee di codice eseguite nella chiamata corrispondente:
  - I nodi interni hanno etichetta 2
  - Le foglie hanno etichetta 1
- Per calcolare T(n) contiamo i nodi dell'albero della ricorsione:
  - numero di foglie : **F(n)**
    - l'albero rappresenta un'espressione aritmetica per calcolare F(n)
  - numero di nodi interni: **F(n)- 1**
    - Il numero di nodi interni di un albero in cui ogni nodo ha due figli è pari al numero di foglie -1

- In totale le linee di codice eseguite sono:

  F(n) + 2 (F(n)-1) = **3F(n)-2**

Osservazioni

`fibonacci2` è un algoritmo lento:

$T(n) \approx F(n) \approx \Phi^n$

Possiamo fare di meglio?

Algoritmo `fibonacci3`

- *fibonacci2* calcola ripetutamente la soluzione dello stesso sottoproblema
- Idea: memorizzare un array le soluzioni dei sottoproblemi

algoritmo `fibonacci3`*(intero n)* $\rightarrow$ *intero*
  sia Fib un array di n interi
  Fib[1] $\leftarrow$ *Fib[2]* $\leftarrow$ *1*
  for i = 3 **to** n **do**
    Fib[i] $\leftarrow$ *Fib[i-1] + Fib[i-2]*
  return *Fib[n]*

Calcolo del tempo di esecuzione

- L'algoritmo `fibonacci3` impiega tempo proporzionale a *n* invece che esponenziale in *n* come `fibonacci2`
- Tempo effettivo richiesto da implementazioni in C dei due algoritmi su piattaforme diverse:

	`fibonacci2`(58)	`fibonacci3`(58)
Pentium IV 1700MHz	15820 sec. ($\simeq$ 4 ore)	0.7 milionesimi di secondo
Pentium III 450MHz	43518 sec. ($\simeq$ 12 ore)	2.4 milionesimi di secondo
PowerPC G4 500MHz	58321 sec. ($\simeq$ 16 ore)	2.8 milionesimi di secondo

Occupazione di memoria

- Il tempo di esecuzione non è l'unica risorsa di calcolo da analizzare
- La quantità di memoria necessaria può essere cruciale:
  - Se un algoritmo è lento, dovremo solo attendere più a lungo per ottenere il risultato, ma se necessita di **più spazio di quello disponibile**, non otterremo mai la soluzione
    - il programma corrispondente non può essere eseguito
  - Nella realtà il tempo per eseguire un programma può dipendere anche dalla quantità di memoria che esso utilizza

Algoritmo `fibonacci4`

`fibonacci3` usa un array di dimensione *n* per memorizzare tutti i valori calcolatii

Idea: basta mantenere solo gli ultimi due valori precedenti

```
algoritmo fibonacci4(intero n) → intero
  a ← b ← 1
  for i = 3 to n do
    c ← a+b
    a ← b
    b ← c
  return b
```

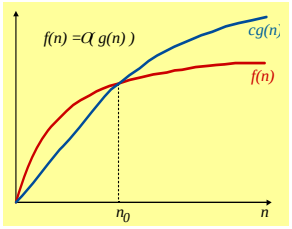

Come valutare T(n)

- Il numero di linee di codice lanciate è una misura molto approssimativa del tempo di esecuzione *T(n)*
  - Se andiamo a capo più spesso, aumentano le linee di codice, ma non il tempo di esecuzione del programma!
  - Per lo stesso programma impaginato diversamente si può avere *T(n)=3n* oppure *T(n)=5n …*
- Si potrebbe usare un preciso modello di calcolo, che stabilisce l'insieme delle istruzioni elementari, ed il loro costo
  - Es., macchina RAM, macchina di Turing, …
  - Ci si può focalizzare sul nr. di operazioni elementari eseguite
- Analisi asintotica: si studia l'ordine di grandezza di *T(n)*
  - ignorando dettagli inessenziali (es. costanti moltiplicative), legate al linguaggio e/o al modello di calcolatore considerati
  - Useremo a questo scopo la notazione asintotica ***O***

Notazione asintotica *O*

- Definizione ("upper bound"):

  $f(n) = O(g(n))$ se $f(n) < c\,g(n)$ per c e n abbastanza grandi



Ad esempio, possiamo rimpiazzare:

- $T(n)=3F_n$ con $T(n)=O(F_n)$
- $T(n)=2n$ e $T(n)=4n$ con $T(n)=O(n)$
- $T(n)= F_n$ con $T(n)=O(2^n)$

- Vantaggi:
  - Più semplice che derivare un'espressione esatta per *T(n)*
  - Si può valutare il numero di istruzioni di alto livello eseguite in funzione di *n*, trascurando termini costanti (… solo "operazioni dominanti")
  - Metodo rapido per verificare l'utilizzabilità pratica di un algoritmo, o per fare un confronto di massima fra algoritmi alternativi

Un ultimo algoritmo per i numeri di Fibonacci

Possiamo sperare di calcolare F_n in tempo inferiore a O(n)?

Potenze ricorsive

- `fibonacci4` non è il miglior algoritmo possibile
- E' possibile dimostrare per induzione la seguente proprietà di matrici:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

- Useremo questa proprietà per progettare un algoritmo più efficiente

Algoritmo `fibonacci5`

algoritmo `fibonacci5`$(intero\ n) \rightarrow intero$

1. $M \leftarrow \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
2. **for** $i = 1$ **to** $n - 1$ **do**
3. $\quad M \leftarrow M \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
4. **return** $M[0][0]$

- Il tempo di esecuzione è ancora O(n)
- Cosa abbiamo guadagnato?

Calcolo di potenze

- Possiamo calcolare la n-esima potenza elevando al quadrato la (n/2)-esima potenza
- Se n è dispari eseguiamo una ulteriore moltiplicazione
- Esempio:

  $3^2=9$ $3^4=(9)^2=81$ $3^8=(81)^2=6561$

Algoritmo fibonacci6

algoritmo fibonacci6(*intero* n) → *intero*

1. $M \leftarrow \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
2. potenzaDiMatrice$(M, n-1)$
3. **return** $M[0][0]$

procedura potenzaDiMatrice(*matrice* M, *intero* n)

4. **if** ($n > 1$) **then**
5. potenzaDiMatrice$(M, n/2)$
6. $M \leftarrow M \cdot M$
7. **if** (n è dispari) **then** $M \leftarrow M \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$

Tempo di esecuzione

- Tutto il tempo è speso in potenzaDiMatrice
  - All'interno della procedura si spende tempo costante
  - Si esegue una chiamata ricorsiva con input n/2
- L'equazione di ricorrenza è pertanto:

$$T(n) = c + T(n/2)$$

- Come risolverla?

Metodo dell'iterazione

$$T(n) \leq c + T(n/2) \leq c + c + T(n/4) = 2c + T(n/2^2)$$

In generale:

$$T(n) \leq kc + T(n/2^k)$$

Per $k=\log_2 n$ si ottiene

$$T(n) \leq c \log_2 n + T(1) = O(\log_2 n)$$

fibonacci6 è esponenzialmente più veloce di fibonacci3!

Riepilogo

	Tempo di esecuzione	*Occupazione di memoria*
fibonacci2	$O(2^n)$	O(n)
fibonacci3	O(n)	O(1)
fibonacci4	O(n)	O(1)
fibonacci5	O(n)	O(1)
fibonacci6	O(log n)	O(log n)

Attenzione:
il valore *n* non è una misura adeguata della dimensione dell'input…