TECNICO SUPERIORE WEB DEVELOPER FULL STACK

#5 - Problemi e algoritmi

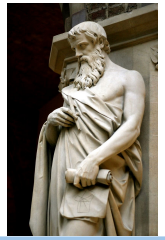
Definizioni

Dati un dominio di input e un dominio di output, un **problema computazionale** è rappresentato dalla relazione matematica che associa un elemento del dominio di input ad ogni elemento del dominio di output.

Dato un problema computazionale, un **algoritmo** è un procedimento effettivo, espresso tramite un insieme di passi elementari ben specificati in un sistema formale di calcolo, che risolve il problema in tempo finito.

Storia degli algoritmi

1850 BC



850 DC

Papiro di Rhind/Ames

Contiene tavole per la moltiplicazione, la divisione in frazioni, ecc.

Algoritmi dei greci

Euclide, per il calcolo del MCD, vari algoritmi geometrici...

Al-Khwarizmi

Matematico uzbeco dal quale deriva il termine “Algoritmo”

La storia degli algoritmi è molto più antica di quella dei calcolatori elettronici, e anche del termine “computer”, che è usato per la prima volta intorno al 1640, per descrivere "colui la cui occupazione è quella di svolgere calcoli aritmetici".

Problema e algoritmo

Il problema:

Dato un insieme S, il minimo di S è l'elemento di S che è minore o uguale ad ogni elemento di S.

$$\min(S) = a \Leftrightarrow \exists\ a \in S : \forall\ b \in S : a \leq b$$

Problema e algoritmo

L’algoritmo (o “un algoritmo”):

Per trovare il minimo di un insieme, confronta ogni elemento con tutti gli altri; l’elemento che è minore di tutti è il minimo.

Descrivere un algoritmo

Per descrivere un algoritmo potremmo scrivere un paragrafo di testo in linguaggio naturale, però è impossibile utilizzarlo per poterci fare dei ragionamenti formali.

Potremmo usare il codice, ma esso dipende dal linguaggio utilizzato e dal livello di astrazione che esso permette.

Differenza di astrazione

C#

```
static int MCD(int a, int b)
{
    while (a != b) {
      if (a < b)
        b = b - a;
      else
        a = a - b;
    }
    return a;
}
```

Assembly

```
main:
   mov x20,36
   mov x21,18
   mov x0,x20
   mov x1,x21
   bl calPGCDmod
   bcs   99f
   mov x2,x0
   mov x0,x20
   mov x1,x21
   bl displayResult
   mov x20,37
   mov x21,15
   mov x0,x20
   mov x1,x21
...
```

Lo pseudo-codice

E’ necessario utilizzare una descrizione il più possibile formale e indipendente dal linguaggio: lo “Pseudo-codice”

```
int min(int[] S, int n)
for i = 1 to n do
    boolean isMin = true
    for j = 1 to n do
        if i ≠ j and S[j] < S[i]
         then
            isMin = false
    if isMin then
        return S[i]
```

Convenzioni dello pseudo-codice

- $a = b$
- $a \leftrightarrow b \equiv$
  $tmp = a; a = b; b = tmp$
- $T[\,]\ A =$ **new** $T[1 \ldots n]$
- $T[\,][\,]\ B =$ **new** $T[1 \ldots n][1 \ldots m]$
- **int**, **float**, **boolean**, **int**
- **and**, **or**, **not**
- $==, \neq, \leq, \geq$
- $+, -, \cdot, /, \lfloor x \rfloor, \lceil x \rceil, \log, x^2, \ldots$
- iif$(condizione, v_1, v_2)$
- **if** *condizione* **then** istruzione
- **if** *condizione* **then** istruzione1
  else istruzione2
- **while** *condizione* **do** istruzione
- **foreach** *elemento* $\in$ *insieme* **do**
  istruzione
- **return**
- % *commento*

Convenzioni dello pseudo-codice

- **for** *indice* = *estremoInf* **to** *estremoSup* **do** istruzione

  int *indice* = *estremoInf*
  while *indice* ≤ *estremoSup* **do**
  istruzione
  indice = *indice* + 1

- **for** *indice* = *estremoSup* **downto** *estremoInf* **do** istruzione

  int *indice* = *estremoSup*
  while *indice* ≥ *estremoInf* **do**
  istruzione
  indice = *indice* − 1

- RETTANGOLO r = **new** RETTANGOLO
- $r.altezza = 10$
- **delete** r
- r = **nil**

RETTANGOLO
int *lunghezza*
int *altezza*

Cos’è la complessità?

E’ l’analisi delle **risorse** impiegate da un algoritmo per risolvere un problema, in funzione della **dimensione** e della **tipologia** dell’input.

Come si valuta un algoritmo?

Risolve il problema in modo efficiente?

- Dobbiamo stabilire come valutare se un programma è efficiente
- Alcuni problemi non possono essere risolti in modo efficiente
- Esistono soluzioni “ottime”: non è possibile essere più efficienti

Come si valuta un algoritmo?

Risolve il problema in modo corretto?

- Dimostrazione matematica, descrizione “informale”
- Alcuni problemi non possono essere risolti
- Alcuni problemi vengono risolti in modo approssimato

Risolvere in modo efficiente

- Non è sufficiente che il nostro algoritmo faccia quello che gli viene richiesto.
- Vogliamo anche che lo faccia in modo efficiente, usando nel modo migliore lo spazio e il tempo.

Tempo e Spazio

Tempo

- L’esecuzione di istruzioni richiede tempo.
- Porta a termine velocemente il task?
- Cosa influenza il tempo di esecuzione?

Spazio

- Le strutture dati occupano spazio.
- Che tipo di strutture dati possono essere usate?
- In che modo la scelta della struttura dati influenza le performance?

Il giusto equilibrio

La soluzione migliore usa un mix equilibrato di tempo e spazio.

- Scegliete delle strutture dati adeguate per modellare il vostro problema;
- Usate degli algoritmi efficienti su queste strutture dati.

Misurare il tempo e lo spazio

Il **tempo** è quello impiegato per completare l'algoritmo, ma come lo misuriamo?

- Con il cronometro?
- Contando il numero di operazioni rilevanti?

Lo **spazio** è la quantità di memoria utilizzata.

A noi interessa la misurazione dell'efficienza in termini di *tempo*.

Misurare con un cronometro

Il tempo effettivamente impiegato per eseguire un algoritmo dipende da troppi parametri:

- bravura del programmatore
- linguaggio di programmazione utilizzato
- codice generato dal compilatore
- processore, memoria (cache, primaria, secondaria)
- sistema operativo, processi attualmente in esecuzione

C’è qualcosa di più “indipendente”?

Misurare il numero di operazioni rilevanti

Cosa si intende con “**rilevanti**”?

Il numero di operazioni che *caratterizzano lo scopo dell’algoritmo*.

- Nel caso della ricerca del minimo, sarà il numero di confronti <
- Nel caso della ricerca di un valore, sarà il numero di confronti ==
- Nel caso di un algoritmo di calcolo, sarà il numero di operazioni aritmetiche…

Esempio: l'algoritmo per il minimo

Contiamo il numero di confronti nel problema del minimo.

```
int min(int[] S, int n)
for i = 1 to n do
    boolean isMin = true
    for j = 1 to n do
        if i ≠ j and S[j] < S[i]
          then
            isMin = false
    if isMin then
        return S[i]
```

Esempio: l'algoritmo per il minimo

E' possibile trovare una soluzione più efficiente?

```
int min(int[ ] S, int n)
% Partial minimum
int min = S[1]
for i = 2 to n do
    if S[i] < min then
        % Update partial minimum
        min = S[i]
return min
```

Risolvere in modo corretto

Per definire la correttezza introduciamo il concetto di **invariante**, ossia una condizione sempre vera in un certo punto del programma.

In particolare ci interessa il concetto di **invariante di ciclo**: una condizione sempre vera all'inizio dell'iterazione di un ciclo.

Invariante di ciclo

Il concetto di invariante di ciclo ci aiuta a dimostrare la correttezza di un algoritmo *iterativo*.

Inizializzazione (caso base): La condizione è vera alla prima iterazione di un ciclo.

Conservazione (passo induttivo): Se la condizione è vera prima di un'iterazione del ciclo, allora rimane vera al termine (quindi prima della successiva iterazione).

Conclusione: Quando il ciclo termina, l'invariante deve rappresentare la "correttezza" dell'algoritmo.

Esempio: il problema del minimo

All'inizio di ogni iterazione del ciclo **for**, la variabile `min` contiene il minimo parziale degli elementi `S[1...i-1]`.

```
int min(int[] S, int n)
int min = S[1]
for i = 2 to n do
    if S[i] < min then
        min = S[i]
return min
```

Altre proprietà degli algoritmi

Semplicità, modularità, manutenibilità, robustezza, …

Secondari in un corso di algoritmi e strutture dati, ma fondamentali nell'ingegneria del software.

Attenzione!

Alcune proprietà hanno un costo aggiuntivo per le prestazioni

- Codice modulare ⇒ pagare il costo della gestione chiamate
- Java bytecode ⇒ paga costo interpretazione

Progettare algoritmi efficienti è un prerequisito!

Complessità degli algoritmi

Misurare la complessità in tempo

Perché stimare la complessità in tempo?

- Per stimare il tempo impiegato per un dato input
- Per stimare il più grande input gestibile in tempi ragionevoli
- Per confrontare l'efficienza di algoritmi diversi
- Per ottimizzare le parti più importanti

La complessità è una funzione "Dimensione dell'input" ⇒ "Tempo"

- Come definire la dimensione dell'input?
- Come misurare il tempo?

Dimensione dell’input

Criterio di costo logaritmico

- La taglia dell’input è il numero di bit necessari per rappresentarlo
- Esempio: moltiplicazione di numeri binari lunghi n bit

Criterio di costo uniforme

- La taglia dell’input è il numero di elementi di cui è costituito
- Esempio: ricerca minimo in un vettore di n elementi

In molti casi:

- possiamo assumere che gli "elementi" siano rappresentati da un numero costante di bit
- le due misure coincidono a meno di una costante moltiplicativa

Definizione di tempo

Tempo ≡ n. istruzioni elementari

Un’istruzione si considera elementare se può essere eseguita in tempo “costante” dal processore.

- `a *= 2?` `// costante => shift`
- `Math.cos(d)?` `// costante => GPU`
- `min(A, n)?` `// dipende da n`

Il modello di calcolo

Rappresentazione astratta di un calcolatore

- Astrazione: deve permettere di nascondere i dettagli
- Realismo: deve riflettere la situazione reale
- Potenza matematica: deve permettere di trarre conclusioni “formali” sul costo

Random Access Machine (RAM)

Memoria:

- Quantità infinita di celle di dimensione finita
- Accesso in tempo costante (indipendente dalla posizione)

Processore (singolo)

- Set di istruzioni elementari simile a quelli reali:
  - somme, sottrazioni, moltiplicazioni, operazioni logiche, etc.
  - istruzioni di controllo (salti, salti condizionati)

Costo delle istruzioni elementari

- Uniforme, ininfluente ai fini della valutazione

Tempo di calcolo `min()`

- Ogni istruzione richiede un tempo costante per essere eseguita
- La costante è potenzialmente diversa da istruzione a istruzione
- Ogni istruzione viene eseguita un certo # di volte, dipendente da `n`
- Avevamo visto che il numero di operazioni rilevanti è n-1

```
ITEM min(ITEM[] A, int n)

ITEM min = A[1]
for i = 2 to n do
    if A[i] < min then
        min = A[i]
return min
```

Qual è il costo di `min()`?

```
ITEM min(ITEM[] A, int n)

ITEM min = A[1]                                  c1   1
for i = 2 to n do                                c2   n
    if A[i] < min then                           c3   n − 1
        min = A[i]                               c4   n − 1
return min                                       c5   1
```

Qual è il costo di `min()`?

```
ITEM min(ITEM[] A, int n)

ITEM min = A[1]
for i = 2 to n do
    if A[i] < min then
        min = A[i]
return min
```

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4(n-1) + c_5$$

$$= c_1 + c_2 n + c_3 n - c_3 + c_4 n - c_4 + c_5$$

$$= (c_2 + c_3 + c_4)n + (c_1 + c_5 - c_3 - c_4)$$

$$= an + b$$

Caso pessimo, ottimo, medio

Caso pessimo:

i dati sui quali l'algoritmo richiede il massimo numero di operazioni.

Caso ottimo:

i dati sui quali l'algoritmo richiede il minor numero di operazioni.

Caso medio:

i dati che richiedono un numero "medio" di operazioni.

Quali sono i casi di $\min$, e quale costo hanno?

Caso pessimo:
L'elemento minimo è l'ultimo.

Caso ottimo:
L'elemento minimo è il primo.

Caso medio:
L'elemento minimo è da qualche parte nel mezzo.

In questo caso, a livello di costo non cambia nulla, perchè l'algoritmo scorre comunque tutta la lista (cambia qualche costante, ma come vedremo è ininfluente).

Qual è il costo della ricerca binaria?

int binarySearch(ITEM[ ] A, ITEM v, **int** i, **int** j)

	Costo	# $(i > j)$	# $(i \leq j)$
if $i > j$ **then**	c_1	1	1
return 0	c_2	1	0
else			
int $m = \lfloor (i+j)/2 \rfloor$	c_3	0	1
if $A[m] = v$ **then**	c_4	0	1
return m	c_5	0	0
else if $A[m] < v$ **then**	c_6	0	1
return binarySearch$(A, v, m+1, j)$	$c_7 + T(\lfloor (n-1)/2 \rfloor)$	0	0/1
else			
return binarySearch$(A, v, i, m-1)$	$c_7 + T(\lfloor n/2 \rfloor)$	0	1/0

Qual è il costo della ricerca binaria?

Assunzioni (Caso pessimo):

- Per semplicità, assumiamo n potenza di 2: $n = 2^k$
- L'elemento cercato non è presente
- Ad ogni passo, scegliamo sempre la parte DX di dimensione n/2

Due casi:

$$T(n) = \begin{cases} c & \text{se } n = 0 \\ T(n/2) + d & \text{se } n \geq 1 \end{cases}$$

$i > j \quad (n = 0) \quad T(n) = c_1 + c_2 = c$

$i \leq j \quad (n \geq 1) \quad T(n) = T(n/2) + (c_1 + c_3 + c_4 + c_6 + c_7) = T(n/2) + d$

Qual è il costo della ricerca binaria?

Soluzione della relazione di ricorrenza tramite espansione

$$T(n) = \begin{cases} c & \text{se } n = 0 \\ T(n/2) + d & \text{se } n \geq 1 \end{cases}$$

$$\begin{aligned} T(n) &= T(n/2) + d \\ &= (T(n/4) + d) + d = T(n/4) + 2d \\ &= T(n/8) + 3d \\ &\dots \\ &= T(1) + kd \quad \leftarrow \textit{dopo k passaggi} \\ &= (T(0) + d) + kd = c + d + kd \\ &= kd + (c+d) \\ &= d\,\mathbf{log}(n) + e \end{aligned}$$

Ordini di complessità

Finora abbiamo analizzato precisamente due algoritmi e abbiamo ottenuto due funzioni di complessità:

Ricerca binaria: $T(n) = d \log(n) + e$ logaritmica

Minimo: $T(n) = an + b$ lineare