Università di Torino – Facoltà di Scienze MFN
Corso di Studi in Informatica
Curriculum SR (Sistemi e Reti)

Algoritmi e Laboratorio a.a. 2006-07
Lezioni

prof. Elio Giovannetti

Parte 6 – Complessità computazionale
di algoritmi e di problemi
versione 27/10/2006

Analisi di algoritmi e di problemi

Tempo e spazio di calcolo

Se P è un programma scritto in un qualche linguaggio di programmazione, in generale sia il tempo di esecuzione T di **P** che lo spazio di memoria S necessario per l'esecuzione di **P** dipenderanno dalla dimensione n dell'input, espressa in unità convenienti (ad esempio, per un programma che opera su un array, la dimensione dell'input sarà il numero di elementi dell'array).

Più astrattamente:

Se **A** è un algoritmo descritto in modo informale, oppure per mezzo di qualche formalismo adatto allo scopo, in generale sia il numero di passi elementari di **A** che lo spazio richiesto da **A** (rispetto ad un opportuno modello di calcolo) dipenderanno dalla dimensione di **A**.

Come si è detto, **vogliamo valutare come tempo e spazio dipendono da n, cioè valutare l'andamento delle funzioni T(n) ed S(n). Ma ...**

in generale **T(n)** ed **S(n)** non sono funzioni !

Caso peggiore, caso migliore, caso medio.

Infatti:

- A parità di dimensione, input diversi possono dar luogo a tempi di calcolo e spazi di occupazione differenti. Esempio: íl tempo impiegato dall'algoritmo di insertion-sort per ordinare un array di una data dimensione n è molto minore nei casi in cui l'array sia già ordinato o quasi ordinato, rispetto ai casi in cui sia ordinato in ordine inverso.
- Le notazioni **T(n)** ed **S(n)** sono quindi ambigue, perché in generale:
  dato un valore di n,
  il valore di T(n) (e di S(n)) non è univocamente determinato.
- Si distingue allora, per ogni n, fra **caso peggiore**, **caso migliore**, e **caso medio**. Cioè si considerano, fra tutti possibili input di dimensione n:
  - quelli che danno luogo al tempo maggiore: sia $T_{worst}(n)$ tale tempo;
  - quelli che danno luogo al tempo minore: sia $T_{best}(n)$ tale tempo;

  Si considera inoltre, per ogni dimensione n:
  - la media $T_{medio}(n)$ dei tempi relativi a tutti i possibili input.

  Analogamente per lo spazio si definiscono $S_{worst}(n)$, $S_{best}(n)$ e $S_{medio}(n)$.

Caso peggiore, caso migliore, caso medio: più formalmente.

- Sia A un algoritmo (descritto in modo informale, o in qualche formalismo inventato allo scopo), oppure un programma scritto in un vero linguaggio di programmazione.
- Sia I un possibile dato (o insieme di dati) in ingresso (o *istanza* di input) per una esecuzione dell'algoritmo (ad esempio, per un algoritmo operante su un array, I è il particolare array passato all'algoritmo).

Siano:

- $t(I)$ = tempo di esecuzione dell'algoritmo per l'input I;
- $s(I)$ = spazio di memoria necessario, in aggiunta allo spazio occupato dal dato stesso in ingresso I, per l'esecuzione dell'algoritmo per l'input I;
- n_I = dimensione dell'input I, espressa in unità convenienti (ad esempio, per un algoritmo che opera su un array, la dimensione dell'input sarà il numero di elementi dell'array)

$t(I)$ ed $s(I)$ di solito dipendono da n_I, ma NON SOLO da n_I.

Caso peggiore, caso migliore, caso medio: più formalmente.

- complessità temporale del caso peggiore:
  $T_{worst}(n) = \max \{t(I) \mid I \text{ ha dimensione } n\}$
- complessità temporale del caso migliore:
  $T_{best}(n) = \min \{t(I) \mid I \text{ ha dimensione } n\}$
- complessità temporale del caso medio:
  $T_{medio}(n) = \text{media } \{t(I) \mid I \text{ ha dimensione } n\}$

Per ogni n, i valori $T_{worst}(n)$, $T_{best}(n)$ e $T_{medio}(n)$ sono univocamente determinati.

$T_{worst}(n)$, $T_{best}(n)$ e $T_{medio}(n)$ sono quindi vere funzioni (di n).

NOTA

Per evitare inutili ripetizioni, le definizioni e le proprietà seguenti sono enunciate solo per la complessità ***temporale***.

Le definizioni e le proprietà per la complessità ***spaziale*** sono tutte perfettamente analoghe (basta sostituire **T** con **S**).

Osservazioni sul caso medio

$T_{medio}(n)$ = media $\{t(I) \mid I$ ha dimensione $n\}$

Che cosa vuol dire ?

Per ogni valore di **n** si considerano tutti i possibili casi di input di quella data dimensione **n**, siano essi in numero **M**:

$$I^n_1, I^n_2, \ldots, I^n_M$$

e si fa la media aritmetica dei tempi di tutti i casi:

$$T_{medio}(n) = (t(I^n_1) + t(I^n_2) + \ldots + t(I^n_M)) / M$$

Spesso, per ragioni inerenti all'uso dell'algoritmo (ad es. perché esso è usato come parte di un algoritmo o di un'applicazione più generale), i possibili casi di input si presentano con probabilità diverse $p_1, p_2, \ldots, p_M$, con $p_1 + p_2 + \ldots + p_M = 1$.

La media deve allora essere una media pesata:

$$T_{medio}(n) = p_1 \cdot t(I^n_1) + p_2 \cdot t(I^n_2) + \ldots + p_M \cdot t(I^n_M)$$

Ancora sul caso medio.

Attenzione:

Anche se assumiamo che, per ciascuna dimensione n, i diversi casi possibili di input siano tutti equiprobabili e quindi la media sia una media aritmetica, la media è la media sui diversi casi, NON sui diversi tempi. Cioè, nella definizione

$$T_{medio}(n) = (t(I^n_1) + t(I^n_2) + \ldots + t(I^n_M)) / M$$

M è il numero dei CASI diversi, NON dei tempi diversi; casi diversi (per la stessa dimensione) potrebbero avere tempi uguali.

Ad esempio: se, per un dato algoritmo, per un certo n ci fossero in tutto 4 input diversi possibili, di cui tre richiedono un tempo t, e uno richiede un tempo t', il tempo del caso medio non è (t + t')/2, ma (t + t + t + t')/4, cioè (3t + t')/4.

Complessità computazionale asintotica.

- Nota Bene: (complessità del) caso peggiore, caso migliore e caso medio non sono tre particolari dati di ingresso (istanze di input), bensì tre funzioni (non decrescenti):

$$T_{worst}(n),\ T_{best}(n),\ T_{medio}(n)$$

- L'espressione matematica precisa di tali funzioni può essere complessa.

- Tuttavia esse hanno di solito andamenti asintotici semplici ben definiti, come quelli elencati nelle slides precedenti.

Delimitazioni superiori e inferiori per gli algoritmi

Dato un algoritmo, abbiamo quindi nove possibili tipi di affermazioni riguardanti la sua complessità, a seconda che si affermi che

$T_x(n)$ è $O(f(n))$ o $\Omega(f(n))$ o $\Theta(f(n))$, con X = best, worst, medio.

Le proprietà espresse da tali affermazioni non sono tutte fra loro indipendenti né ugualmente interessanti. In particolare, per ogni algoritmo si ha:

$T_{worst}(n)$ è $O(f(n))$ → $T_{medio}(n)$ è $O(f(n))$ → $T_{best}(n)$ è $O(f(n))$

Cioè: **se il tempo di calcolo cresce non più di f(n) nel caso peggiore, ovviamente cresce non più di f(n) anche nei casi migliore e medio.**

Simmetricamente:

$T_{best}(n)$ è $\Omega(f(n))$ → $T_{medio}(n)$ è $\Omega(f(n))$ → $T_{worst}(n)$ è $\Omega(f(n))$

Cioè: **se il tempo di calcolo cresce almeno come f(n) nel caso migliore, ovviamente cresce almeno come f(n) anche nei casi medio e peggiore.**

Delimitazioni superiori e inferiori per gli algoritmi

Sono quindi giustificate le seguenti definizioni:

- Un algoritmo **ha complessità O(f(n))** se è $T_{worst}(n) = O(f(n))$.

  In tal caso f(n) è infatti una **delimitazione superiore** del tempo di calcolo per qualsiasi input: si ha la garanzia che al crescere di n **il tempo di calcolo non cresce di più di f(n), qualunque sia l'input**.

- Un algoritmo **ha complessità Ω(f(n))** se è $T_{best}(n) = \Omega(f(n))$.

  In tal caso f(n) è infatti una **delimitazione inferiore** del tempo di calcolo per qualsiasi input: si ha purtroppo la certezza che al crescere di n **il tempo di calcolo cresce almeno come f(n), qualunque sia l'input**.

- Un algoritmo **ha complessità Θ(f(n))** se ha complessità $O(f(n))$ e $\Omega(f(n))$.

  In tal caso f(n) è infatti una **delimitazione sia inferiore che superiore** del tempo di calcolo per qualsiasi input.

Altri modi di esprimersi

Equivalentemente, diciamo che:

- il tempo di esecuzione di un algoritmo è $O(f(n))$ se $T_{worst}(n)$ è $O(f(n))$,
- il tempo di esecuzione di un algoritmo è $\Omega(f(n))$ se $T_{best}(n)$ è $\Omega(f(n))$,
- il tempo di esecuzione di un algoritmo è $\Theta(f(n))$ se è sia $O(f(n))$ che $\Omega(f(n))$

benché il "tempo di esecuzione" senza specificazioni non sia una funzione (solo) di n e quindi a rigore non si potrebbe parlare di O, Ω e Θ.

In qualunque modo ci esprimiamo, abbiamo evidentemente esteso il significato di O, Ω e Θ in modo naturale.

Esempio: insertion-sort

- Il tempo di esecuzione nel caso peggiore è $\Theta(n^2)$.
- Il tempo di esecuzione nel caso migliore è $\Theta(n)$.

Quindi:

- Il tempo di esecuzione è $O(n^2)$,
  perché è $\Theta(n^2)$ – e quindi anche $O(n^2)$ – nel caso peggiore.
- Il tempo di esecuzione è $\Omega(n)$,
  perché è $\Theta(n)$ – e quindi anche $O(n)$ – nel caso migliore.
- Il tempo di esecuzione non è né $\Theta(n^2)$ né $\Theta(n)$:
  per ogni dimensione n vi sono sia input per cui il tempo è proporzionale a n, sia input per cui è proporzionale a n^2.

Caso peggiore, caso migliore, caso medio: osservazioni.

- **caso migliore**: di solito è scarsamente interessante, poco più di una curiosità, poiché riguarda input molto particolari e "improbabili"; un'eccezione è l'**insertion sort**, il cui caso migliore (sequenza ordinata) è un caso significativo;

- **caso medio**: è significativo solo se la distribuzione delle probabilità riflette la situazione reale di uso dell'algoritmo: una determinazione sensata di tale distribuzione può essere difficile; inoltre in applicazioni critiche l'efficienza nel caso medio può non essere una garanzia sufficiente; un'eccezione è il **quicksort**, in cui il caso medio è quello interessante;

- **caso peggiore**: è la complessità che si studia di solito, poiché è l'unica che fornisce la garanzia che in ogni caso il tempo di esecuzione non sarà maggiore di un certo tempo prevedibile.

Complessità di problemi.

Delimitazione superiore (upper bound) di un problema.

Un problema ha complessità O(f(n)) nel caso peggiore/migliore/medio se ***esiste*** un algoritmo di complessità del caso peggiore/migliore/medio O(f(n)) che lo risolve.
cioè
se si sa come risolverlo in un tempo che cresca non più di f(n).

Delimitazione inferiore (lower bound) di un problema.

Un problema ha complessità Ω(f(n)) nel caso peggiore/migliore/medio se ***tutti*** i possibili algoritmi risolventi (anche quelli non ancora inventati !) hanno complessità del caso peggiore/migliore/medio Ω(f(n)),
ossia se qualunque algoritmo che risolva il problema deve avere complessità del caso peggiore/migliore/medio Ω(f(n)).
cioè
se è impossibile risolverlo in un tempo che cresca meno di f(n).

Complessità di problemi.

Per brevità, nel seguito scriviamo *complessità* invece di ripetere ogni volta l'espressione *complessità del caso peggiore/migliore/medio*.

Nota Bene:

- Per trovare una delimitazione superiore **f(n)** alla complessità di un problema è sufficiente (e necessario) trovare **UN algoritmo** che risolve il problema in un tempo **O(f(n))**.

- Per trovare una delimitazione inferiore **g(n)** è necessario **dimostrare matematicamente** che qualunque possibile algoritmo, per risolvere il problema, deve impiegare un tempo che cresce almeno come **g(n)**.

  NON BASTA che tutti gli algoritmi conosciuti che risolvono il problema abbiano complessità **Ω(g(n))** !

Status storico dei problemi.

Un problema algoritmico, in un dato momento storico,
può essere:

- **aperto**, cioè presentare un ***gap algoritmico***;
- **chiuso**.

Come sarà ovvio in base alle definizioni di "aperto" e "chiuso",
un problema aperto può venire chiuso,
ma un problema chiuso non può ridiventare aperto !

Problema algoritmicamente chiuso

È un problema di cui si conoscono una delimitazione superiore e una inferiore coincidenti. Cioè è un problema per cui:

- esiste almeno un algoritmo risolvente di complessità $O(f(n))$;
- si è dimostrato che qualunque algoritmo risolvente deve avere una complessità che è $\Omega(f(n))$, cioè che non può esistere un algoritmo di complessità inferiore a $\Omega(f(n))$.

Si è quindi dimostrato che non si possono trovare algoritmi asintoticamente migliori, l'algoritmo risolvente è *ottimo*: saranno possibili solo miglioramenti marginali, ad es. per una costante additiva o moltiplicativa.

Problema con gap algoritmico

È un problema per cui (tutte le) delimitazioni inferiori e superiori differiscono. Cioè è un problema per cui:

- il miglior algoritmo risolvente conosciuto ha complessità $O(f(n))$;
- si è dimostrato che qualunque algoritmo risolvente deve avere complessità $\Omega(g(n))$, con $g \neq f$.

Cioè:

- si sa risolvere il problema in un tempo $f(n)$;
- si sa che non lo si può risolvere in un tempo migliore di $g(n)$ (dove ovviamente $g(n)$ "cresce meno" di $f(n)$.

Esempio:

esiste un algoritmo risolvente esponenziale e si è dimostrato che nessun algoritmo risolvente può essere meglio che lineare.

Come chiudere il gap ?

Il gap algoritmico è la "differenza" fra le due delimitazioni.
Esso può venire chiuso, o almeno "ridotto", in due modi:

- dal di sopra: si riesce a trovare un algoritmo migliore di quelli fino ad allora esistenti, abbassando così il limite superiore; se si trova un algoritmo di complessità coincidente con il limite inferiore, tale algoritmo è ottimo e il problema è chuso;
- dal di sotto: si riesce a dimostrare un limite inferiore più alto; se si riesce a dimostrare che nessun algoritmo può essere asintoticamente meglio del migliore esistente, si è dimostrato che l'algoritmo è ottimo e il problema è chiuso.

I due modi non sono mutuamente esclusivi; può succedere che:

- si riesca a trovare un algoritmo migliore,
- e contemporaneamente si riesca a dimostrare un limite inferiore più alto.

Un gap aperto (di una classe famosa)

Il problema della soddisfacibilità di una formula booleana:

- esiste un algoritmo risovente di complessità esponenziale (l'algoritmo banale che prova tutte le possibili combinazioni di valori di verità);
- è facile dimostrare che qualunque algoritmo risolvente deve avere complessità almeno polinomiale;

ma:

- nessuno è mai riuscito a trovare un algoritmo risolvente che sia asintoticamente più efficiente (cioè polinomiale);
- nessuno è ancora riuscito a dimostrare che un tale algoritmo (cioè polinomiale) non può esistere, anche se si ha il fondato sospetto che sia così.

Con un gioco di parole potremmo quindi dire che è

un problema proBabilmente intrattabile,
ma non proVabilmente intrattabile !

Il problema della soddisfacibiltà booleana è un caso particolare di un problema più generale di informatica teorica classificato come uno dei "Problemi matematici del Millennio", per ognuno dei quali è in palio un premio di un milione di dollari.

Buona fortuna !

Un problema algoritmico insolubile

Si definisca una procedura

```
static boolean stops(File prog, File input)
```

la quale, preso in input un file costituente un programma Java e un file rappresentante un input legale per tale programma, esamini il programma e restituisca true o false a seconda che il programma eseguito con quell'input termini oppure no.

Una tale procedura non può esistere !

Dimostrazione

Se la procedura static boolean stops(File prog, File input) esistesse:

- Grazie ad essa si potrebbe definire la procedura
  static boolean selfStops(File prog) {
    return stops(prog, prog);
  } la quale, preso in input un programma, restituisce true se il programma eseguito con input se stesso termina (eventualmente con un errore), false altrimenti.
- Utilizzando a sua volta la precedente, si potrebbe definire la procedura:
  static void strange(File prog) {
    if(selfStops(prog) then while(true);
    else System.out.println("ho finito");
  } la quale, preso in input un programma *prog*, non termina se *prog* applicato a *prog* termina, e invece termina se *prog* applicato a *prog* non termina.
- Allora, se applichiamo il programma *strange* ad un file contenente se stesso, sostituendo *prog* con *strange* otteniamo: *strange* applicato a *strange* non termina se *strange* applicato a *strange* termina, e viceversa;

  cioè non termina se termina, e termina se non termina ! Contraddizione !

Riassumendo

Vi sono diversi generi di problemi algoritmici:

- problemi risolti da un algoritmo efficiente (logaritmico, lineare, pseudo-lineare) e per i quali si è dimostrato che non possono esistere algoritmi asintoticamente migliori (problemi "facili" chiusi); esempio: il problema dell'ordinamento;
- problemi risolti da un algoritmo efficiente o comunque polinomiale, che però non si sa se sia quello asintoticamente migliore (problemi trattabili, con gap algoritmico); esempio: il prodotto di matrici;
- problemi "presumibilmente" intrattabili: problemi per i quali gli unici algoritmi risolventi sono esponenziali, e per i quali si sospetta fortemente – ma non si è dimostrato – che non esistano algoritmi migliori; esempi: soddisfacibilità booleana, problema del commesso viaggiatore;
- problemi dimostrabilmente intrattabili: problemi per i quali gli unici algoritmi risolventi sono esponenziali, e per i quali si è dimostrato che non possono esistere algoritmi migliori; esempi: le torri di Hanoi, il problema dei blocchi stradali.
- problemi dimostrabilmente insolubili: problemi per i quali si è dimostrato che non possono esistere algoritmi risolventi; esempi: il problema della terminazione, il problema dell'equivalenza fra programmi.

Nota: titoli di alcuni capitoli del libro divulgativo di D. Harel

Algorithmics

The Spirit of Computing

- The Correctness of Algorithms: Getting it done right.
- The Efficiency of Algorithms: Getting it done cheaply.
- Intractability: You can't always get it done cheaply.
- Noncomputability: Sometimes you can't get it done at all !