Introduzione alla Complessità Computazionale

Sommario

- Motivazioni
- Efficienza di un algoritmo
- Modelli di costo
- Complessità asintotica e ordini di grandezza
- Calcolo della complessità
- Classi di complessità

Motivazioni (1)

"Fino a poco tempo fa, i matematici teorici consideravano un problema risolto se esisteva un algoritmo per risolverlo; il procedimento di esecuzione era di importanza secondaria. Tuttavia, c'è una grande differenza tra il sapere che è possibile fare qualcosa e il farlo. Questo atteggiamento sta cambiando rapidamente, grazie ai progressi della tecnologia del computer. Adesso, è importantissimo trovare metodi di soluzione che siano pratici per il calcolo.
La teoria della complessità studia i vari algoritmi e la loro relativa efficienza computazionale. È una teoria giovane e in pieno sviluppo, che sta motivando nuove direzioni nella matematica e nello stesso tempo trova applicazioni concrete quali quello fondamentale della sicurezza e identificazione dei dati."

E. Bombieri, Medaglia Fields 1974, in *La matematica nella società di oggi*, Bollettino UMI, Aprile 2001

Motivazioni (2)

- Un problema può essere spesso risolto utilizzando algoritmi diversi
  - Come scegliere il *migliore*?
- La *bontà* (efficienza) di un algoritmo si misura in base alla quantità minima di risorse sufficienti per il calcolo: Tempo / Spazio

Misurare il tempo di calcolo (1)

- La misurazione del tempo di calcolo di un algoritmo in base al tempo cronologico (ad es. in μsec) non è sensato
- Cambia con
  - i dati di input
  - il linguaggio di programmazione
  - la qualità della traduzione in sequenze di bit
  - la velocità dell'elaboratore
- Il miglioramento della tecnologia non riduce significativamente il tempo di esecuzione

Misurare il tempo di calcolo (2)

- Necessità di una misura astratta
- Necessità
  - di prescindere dallo specifico sistema di calcolo
  - di concentrarsi sulle caratteristiche specifiche del metodo

Misurare il tempo di calcolo (3)

- La complessità dei problemi da risolvere dipende dalla dimensione dei dati in ingresso
- Tempo di Calcolo espresso come numero complessivo di operazioni elementari in funzione della dimensione n dei dati in ingresso

Misurare il tempo di calcolo (4)

- Operazioni elementari:
  - aritmetiche/logiche/confronto/assegnamento
- Il numero di operazioni considerate è valutato nel **caso peggiore**
  - nel caso di dati in ingresso più sfavorevoli tra tutti quelli di dimensione n

Esempi (1)

- Trovare il minimo in un insieme di n numeri
- Soluzione:
  - Primo numero candidato ad essere il minimo
  - confronto il candidato con tutti gli altri elementi e se trovo un elemento più piccolo lo faccio diventare il nuovo candidato
  - al termine dei confronti, il candidato corrente è sicuramente il minimo
- Complessivamente n confronti:
  - L'efficienza dell'algoritmo è quindi direttamente proporzionale alla dimensione n dell'input

Esempi (2)

- Ordinare in ordine crescente un insieme di n numeri $\{x_1, x_2, \ldots, x_n\}$
- Soluzione:
  - per i = 1, 2,…, n ripeti le seguenti operazioni:
    - trova il minimo nel sottoinsieme $\{x_i, x_{i+1}, \ldots, x_n\}$
    - scambialo con x_i
- Per n volte bisogna trovare il minimo su insiemi sempre più piccoli
- Si eseguono n + (n-1) + (n-2) + … + 2 + 1, cioè circa **n^2 operazioni**

Efficienza di un algoritmo

- Si esprime come funzione f(n) della variabile n, che esprime il numero di operazioni compiute per un problema di dimensione n
- Rappresenta la complessità computazionale dell'algoritmo
  - Se A e B sono due algoritmi che risolvono lo stesso problema e se $f_A(n)$ e $f_B(n)$ rappresentano la complessità dei due algoritmi allora A è migliore di B se, al crescere di n, risulta $f_A(n) \leq f_B(n)$

Efficienza di un algoritmo - Esempio

- Siano due algoritmi diversi per ordinare *n* interi
  - Il primo esegue n^2 istruzioni
  - Il secondo $n * \log n$ istruzioni
- Supponiamo che l'esecuzione di un'istruzione avvenga in un µsec (10^{-6} sec), allora si ha

	n=10	n=10000	$n=10^6$
n^2 istruzioni	0,1 msec	100 sec (~1,5 min)	10^6 sec (~12 gg)
$n * \log n$ istruzioni	23 µsec	92 msec	13,8 sec

Modello di Costo (1)

- Per definire un modello di costo di un algoritmo è necessario stabilire
  - la dimensione dell'input
  - la istruzione di costo unitario (passo base)
  - la complessità
    - nel caso migliore
    - nel caso peggiore
    - nel caso medio

Modello di Costo (2)

- A seconda del problema, per dimensione dell'input si possono intendere cose diverse:
  - La grandezza del numero (ad es. nei problemi di calcolo)
  - Quanti elementi sono in ingresso (es: ordinamento)
  - Quanti bit compongono un numero
- Indipendentemente dal tipo di dati indichiamo con n la dimensione dell'input

Modello di Costo (3)

- L'operazione di costo unitario è un'operazione la cui esecuzione non dipende dai valori e dai tipi delle variabili
  - Assegnamento e operazioni aritmetiche di base
  - Accesso ad un elemento qualsiasi di un vettore
  - Valutazione di un'espressione booleana
  - Istruzioni di I/O

Calcolo di complessità in numero di passi base (1)

```
i = 0;
while(i < n)
        i = i+1;
```

- Assegnamento esterno (i=0) → 1
- Numero di test della condizione →n+1
- Assegnamenti interni →1*n
- Totale 2*n+2

Calcolo di complessità in numero di passi base (2)

```
i = 0;
while(i < n){
        i = i+1;
        j = j*3 + 42;
}
```

- Assegnamento esterno (i=0) → 1
- Numero di test della condizione →n+1
- Assegnamenti interni →2*n
- Totale 3*n+2

Calcolo di complessità in numero di passi base (3)

```
i = 0;
while(i <2*n){
        i = i+1;
        j = j*3 + 4367;
}
```

- Assegnamento esterno (i=0) → 1
- Numero di test della condizione →2*n+1
- Assegnamenti interni →2*(2*n)
- Totale 6*n+2

Complessità Asintotica (1)

- Supponiamo:
  - di avere sei algoritmi con diversa complessità
  - un passo base venga eseguito in un µsec (10^{-6} sec)
- Allora:

Complessità	n = 10	n = 100	n = 1000	n = 10^6
n + 5	15 µsec	10^{-4} sec	10^{-3} sec	1 sec
2*n	$2*10^{-5}$ sec	$2*10^{-4}$ sec	$2*10^{-3}$ sec	2 sec
n^2	10^{-4} sec	10^{-2} sec	1 sec	10^6 (~12 giorni)
$n^2 + n$	10^{-4} sec	10^{-2} sec	1 sec	10^6 (~12 giorni)
n^3	10^{-3} sec	1 sec	10^5 (~1 giorno)	10^{12} (~300 secoli)
2^n	10^{-3} sec	~$4*10^{14}$ secoli	~$3*10^{287}$ secoli	~$3*10^{301016}$ secoli

Complessità Asintotica (2)

- Ha senso discutere della complessità di un algoritmo rispetto al suo comportamento per valori grandi di n (comportamento asintotico)
- Affermare che un algoritmo ha un certo andamento asintotico significa dire che per qualunque configurazione degli input si ha quel tipo di comportamento
  - Non è corretto valutare il comportamento asintotico solo nel caso migliore/peggiore
  - Si deve analizzare l'andamento per tutti i casi possibili

Ordini di Grandezza

- Per esprimere l'andamento asintotico della complessità si usa l'ordine di grandezza
- Normalmente si fa riferimento a tre diverse notazioni: O, Ω, Θ

O(f(n)) - Definizione

- $O(f(n)) = \{g(n) : \exists\, c > 0, n_0 > 0$ tali che $\forall\, n > n_0$ $0 \leq g(n) \leq c\, f(n)\}$
  - A partire da una certa dimensione n_0 dei dati di input, la funzione f(n) maggiora la funzione g(n)
  - La funzione f(n) rappresenta un limite superiore per la g(n)
- Tale limite non è stretto
  - Ad es. se $f(n) \in O(n^2)$, allora la f(n) da un certo punto in poi è maggiorata da n^2, ma allora anche n^3 maggiora la f(n), quindi si ha anche che $f(n) \in O(n^3)$

Ω(f(n)) - Θ(f(n)) Definizioni

- $\Omega(f(n)) = \{g(n) : \exists\, c > 0, n_0 > 0$ tali che $cf(n) \leq g(n), \forall\, n > n_0\}$
  - Definisce il limite inferiore
- $\Theta(f(n)) = \{g(n) : \exists\, c_1, c_2 > 0$ tali che $c_1 f(n) \leq g(n) \leq c_2 f(n), \forall\, n > n_0\}$
  - È la classe delle funzioni che hanno lostesso andamento asintotico della f

Programmi Strutturati

- Nel calcolo della complessità di un programma strutturato si deve:
  - Calcolare la complessità di ogni funzione
  - Per ogni chiamata a funzione, aggiungere la complessità della stessa al costo globale del programma

Calcolo della Complessità Computazionale (1)

- In riferimento alle strutture di controllo, è possibile procedere al calcolo della complessità computazionale degli algoritmi
- Istruzioni singole
  - si assume abbiano complessità costante, indicata con O(1)
- Blocco sequenziale: è costituito dalla sequenza di istruzioni singole I1; I2; … In
  - La complessità del blocco sarà la complessità massima delle istruzioni che lo compongono

Calcolo della Complessità Computazionale (2)

- Selezione: If (cond) {blocco_then} else {blocco_else}
  - Sia f_cond la complessità della valutazione della condizione (costante); f_then la complessità di blocco_then; f_else la complessità di blocco_else
  - La complessità della selezione è f_if=O(max(f_cond + f_then, f_cond + f_else))
- Analogo ragionamento vale per il case

Calcolo della Complessità Computazionale (3)

- Iterazione: While(cond) {blocco}
  - È la complessità del blocco, moltiplicata per il numero di volte in cui è eseguito, a cui aggiungere la complessità della valutazione della condizione
  - La complessità del while è f_while=O(f_cond + k*f_blocco), dove k è il numero massimo di volte che il ciclo viene iterato
- Analoghi ragionamenti per do-while e per for

Calcolo della Complessità Computazionale (4)

- Chiamata di funzioni
  - È la somma della complessità dell'esecuzione della funzione più la complessità della istruzione di chiamata

Calcolo della Complessità Computazionale (4)

- Chiamate ricorsive
  - La complessità *C(n)* di una funzione ricorsiva R è data dal contributo:
    - della complessità di tutte le istruzioni in R che NON sono ricorsive
    - della complessità *C(k)* che deriva dalle chiamate del passo ricorsivo

Classi di Complessità per Problemi (1)

- Calcolandone la complessità computazionale, è possibile stabilire il grado di efficienza di un algoritmo
  - tanto più basso è il numero di operazioni eseguite per calcolare la soluzione di un problema, tanto maggiore sarà l'efficienza dell'algoritmo
- In questo modo è possibile confrontare fra loro algoritmi differenti che consentono di risolvere il medesimo problema

Classi di Complessità per Problemi (2)

- È utile estendere il concetto di complessità dagli algoritmi ai problemi
- Domanda: è possibile identificare la complessità computazionale che caratterizza intrinsecamente un determinato problema, a prescindere dalla complessità di uno specifico algoritmo risolutivo per il problema stesso?

Classi di Complessità per Problemi (3)

- Risposta: definiamo complessità computazionale di un problema la complessità dell'algoritmo più efficiente che lo risolve
  - non è necessario esibire un simile algoritmo, è sufficiente dimostrare che esiste e che tale è la sua complessità computazionale.
- Una volta definita la complessità di un problema, diremo che ogni algoritmo caratterizzato dalla medesima complessità è un algoritmo *ottimo* per la soluzione di tale problema

Classi di Complessità per Problemi (4)

- Quindi è possibile associare ogni problema Turing-calcolabile ad una specifica classe di complessità computazionale

Classi P e NP (1)

- Indichiamo con P la classe di complessità dei **problemi polinomiali**
  - cioè l'insieme di tutti i problemi che ammettono un algoritmo risolutivo deterministico con complessità polinomiale, cioè espressa come polinomio della dimensione dell'input

Classi P e NP (2)

- Indichiamo con NP la classe di complessità dei **problemi polinomiali non deterministici**
  - cioè l'insieme di tutti i problemi che ammettono un algoritmo risolutivo non deterministico con complessità polinomiale
  - NON è la classe dei problemi non polinomiali

Classi P e NP (3)

- Un algoritmo deterministico è un caso particolare della classe degli algoritmi non deterministici
- Quindi la classe P è contenuta nella classe NP
  - Si tratta di un sottoinsieme proprio? $P \subseteq NP$ oppure $P \subset NP$?