Università degli Studi di Verona

CORSO DI LAUREA IN INFORMATICA

Algoritmi

Federico Brutti
federico.brutti@studenti.univr.it

Inserire citazione inerente alla materia

Indice

Capitolo 1

Complessità degli algoritmi

Il corso di algoritmi propone di fornire una comprensione su come descrivere correttamente una procedura in termini matematici. Daremo delucidazioni sulla complessità, il modus operandi per una scelta consapevole e studieremo anche alcuni algoritmi notevolu utili per l'organizzazione di strutture dati.

Anzitutto, definiamo **algoritmo** una descrizione di una procedura di calcolo, la quale deve essere riproducibile e rigorosa abbastanza da non risultare ambigua. Degli algoritmi studieremo la loro **complessità**, la misura del loro tempo di esecuzione, per poi confrontare quale si adatta meglio allo scopo preso in esame. Le metodologie saranno affrontate più avanti.

1.1 Concetto di complessità

La complessità di un algoritmo indica il tempo, le risorse ed eventualmente anche quanti processori usa per essere eseguito. Non è un valore preciso, poiché varia a seconda di quanto è grande il programma e la relativa quantità di dati da elaborare; infatti si descrive tramite una funzione lineare. Quest'ultima è il prodotto che eventualmente si dovrà mostrare al richiedente ed è buona prassi cercare di renderla il più semplice e compatta possibile. Essendo che abbiamo spazio di manovra sulla descrizione, sta a noi scegliere quanto essere precisi nella formula.

L'iter di lavoro generale per la descrizione della funzione di complessità segue due passaggi:

- **Dimostrazione caso pessimo** $O(m)$: L'istanza dove peggio di così l'algoritmo non può fare. Bisogna trovare il limite superiore della funzione.
- **Dimostrazione caso ottimo** $\Omega(n)$: Il miglior scenario dove l'algoritmo svolge le operazioni nel minor tempo possibile. Qui è necessario fare delle assunzioni sul caso

peggiore non sarà sempre possibile ottenere il caso perfetto, tuttavia se coincide con il limite inferiore della funzione, si dice che la stima è perfetta.

Esempio 1. ***Ricerca di un elemento in un array*** $T(n) = c * n$
Il tempo di esecuzione di questo algoritmo è direttamente proporzionale alla dimensione dell'array, poiché ad ogni elemento aggiunto bisognerà controllare anche quelli. c è un valore costante, mentre n è la dimensione, rappresenta le operazioni da eseguire.

Noi sappiamo che la ricerca della lunghezza dell'array è fatta a tempo costante, perché è un'operazione uguale indipendentemente dalla mole di dati. Ogni assegnazione è un'operazione. Nel ciclo while si effettua un test, anch'esso equivalente ad un'operazione, ma ripetuta tante volte finché non si trova l'elemento. Otteniamo quindi i valori:

- *a = assegnazione del valore della lunghezza in n*
- *1 = assegnazione dell'indice*
- *d = ciclo while che effettua il testing*
- *b = incremento indice*

Scrivendo tutto, otteniamo la funzione: $a+1+d+n(b+d)$. *Tuttavia questa è una scrittura non chiara, quello che dobbiamo fare noi infatti si chiama* ***analisi asintotica****, e ci consente di considerare solamente gli ordini di grandezza, ignorando le costanti. Sapendo questo, rimuoviamo i termini di ordine inferiore per ottenere la soluzione effettiva:* $n(b+d)$.

1.2 Notazione asintotica

Abbiamo visto un caso semplice, ma è necessario chiarire come viene determinato l'ordine di grandezza di una funzione. Anzitutto consideriamo che non ne esiste uno specifico, possiamo tuttavia definirli come uno diverso dall'altro. Diciamo infatti che una funzione f appartiene ad un dominio di grandezza di g, dove, a meno di costanti, quest'ultima cresce non meno rapidamente della prima. Formalmente:

$$f \in O(g) \implies [(\exists c > 0) \wedge (\exists \overline{n})].\forall n > \overline{n} \implies f(n) \leq cg(n).$$

Quindi diciamo che la funzione f appartiene a **O grande** di g e la funzione di complessità indica il caso peggiore. Ciò implica l'esistenza di una costante $c > 0$ e di un valore $\overline{n}$ tali per cui ogni altro n è maggiore stretto di $\overline{n}$. Ne consegue che la funzione $f(n)$ crescerà più lentamente o allo stesso modo di $cg(n)$.

Esempio 2. *Sia la funzione* $T(n) = 2x \in O(5x + 7)$. *Bisogna trovare una costante c tale per cui valga la disequazione:* $2x \leq c(5x + 7)$. *noterai che andrà bene ogni c.*

Esempio 3. *Sia la funzione* $T(n) = 5x + 7 \in O(2x)$. *Trovare un* c *tale per cui valga* $5x + 7 \leq c(2x)$.
Stiamo lavorando con un'analisi asintotica, quindi possiamo rimuovere ogni costante, otterremo quindi: $5x \leq c(2x)$. *Numero per superare* 5*? Scegliamo* $c = 3$. *Risolvendola, noterai che la relazione risulta corretta, poiché* $\overline{n} = 7$.

Per la dimostrazione delle equazioni nei diversi ordini di grandezza è possibile usare il **metodo di sostituzione**, preso direttamente da logica matematica. Fondamentalmente bisogna arrivare alla tesi tramite passaggi logici.

Esempio 4. *Supponiamo che la seguente formula sia vera:* $f_1 \in O(g_1), f_2 \in O(g_2) \implies f_1 + f_2 \in O(g_1 + g_2)$
Visto e considerato che stiamo ragionando al netto di costanti, bisogna seguire i seguenti passaggi generali, applicati al caso in esame:

1. *Scrivere i dati in base alla loro definizione, quindi:*

$$f_1 \leq c_1 g_1(n); f_2 \leq c_2 g_2(n)$$

2. *Identifica ciò che è necessario per la dimostrazione; in questo caso la costante* c *ed il risultato della disequazione* $\overline{n}$*:*

$$c = c_1 + c_2; \overline{n} = \overline{n_1} + \overline{n_2}$$

3. *Verifica la tesi:*

$$\begin{aligned} T(n) &= f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n) \\ &= f_1(n) + f_2(n) \leq c(g_1(n) + g_2(n)) \\ &= f_1(n) + f_2(n) \leq c(g_1 + g_2)(n) \implies f_1(n) + f_2(n) \leq \in O(g_1 + g_2) \end{aligned} \tag{1.1}$$

Dove ragionare sul caso peggiore di un algoritmo è necessario, non è sufficiente per dare una visione completa della procedura. Infatti possiamo andare a vedere anche il caso ottimo, dove diremo che una funzione $f \in \Omega(g)$, quindi che asintoticamente, al netto di costanti f non sta sotto g. Formalmente:

$$\exists(c > 0 \wedge \overline{n}) \forall n > \overline{n} \implies f(n) \geq cg(n)$$

Anche questo concetto è dimostrabile tramite sostituzione logica:

Esempio 5. *Sia una funzione* $f \in O(g) \iff g \in \Omega(f)$. *Quindi bisogna dimostrare che* f *sta sotto a* g *se e solo se* g *sta sopra* f.

1. ***Dimostra che vale*** $a \implies b$ ***e che*** $b \implies a$ ***scrivendo la definizione effettiva***
$$\exists c \exists \overline{n} \forall n > \overline{n} f(n) \leq cg(n)$$

2. ***Ora ci serve una funzione*** $g(n) \geq c'f(n)$*:*
$$cg(n) \geq f(n) \rightarrow g(n) \geq f(n)/c$$
Per rendere il tutto più leggibile, possiamo scrivere $c' = \frac{1}{c}$ *e per renderlo vero diremo che* $\overline{n'} = \overline{n}$*.*

Il succo del concetto è che per definire correttamente la complessità di un algoritmo, bisogna dimostrare sia il limite superiore, ovvero $O(n)$, che il limite inferiore $\Omega(n)$, con lo scopo di dare una visuale sul caso peggiore e migliore ottenibili. La scelta di quanto restringere la definizione è arbitraria e dipende dall'ideatore o dalla stupidità del richiedente. In ogni caso, va esposto con una funzione chiara e coincisa.

1.3 Equazioni di ricorrenza

Come visto nel corso di programmazione, ci sono due modi per la costruzione di algoritmi; il metodo **iterativo** ed il metodo **ricorsivo**. Da un punto di vista analitico si tende a preferire una scrittura ricorsiva piuttosto che iterativa poiché la prima risulta più leggibile e meno complessa da scrivere.

Ogni algoritmo è rappresentato con le **equazioni di ricorrenza**, le quali possono essere risolte con questi due paradigmi appena menzionati. Queste sono generalmente notate come:
$$T(n) = \begin{cases} c; n < \overline{n} \\ f(n-1, ..., 1); n \geq \overline{n} \end{cases}$$
Ci interessa esclusivamente l'ordine di grandezza, quindi prendiamo solo il caso ricorsivo al posto di quello base. Essendo inoltre $\overline{n}$ un valore arbitrario, possiamo rendere il caso base molto grande o piccolo, quindi abbiamo spazio di manovra. In ogni caso, trovata l'intersezione fra il limite superiore ed inferiore, indichiamo che abbiamo capito perfettamente l'ordine di grandezza ed è segnalato con $\theta(n) = O(n) \cap \Omega(n)$.

Esempio 6. ***Metodo iterativo***
Sia la funzione $T(n) = 1+T(n-1)$*. Si parte dalla* $T(n)$ *fino ad esaurire tutte le iterazioni in un certo indice* i*.*

$$\begin{aligned} T(n) &= 1 + T(n-1) \\ &= 1 + 1 + T(n-2) \\ &= 1 + 1 + 1 + T(n-3) \\ &= 1 + ... + 1 + T(n-i) \end{aligned} \tag{1.2}$$

Dopo un certo numero di passaggi, sarai arrivato al caso base $T(1)$*, dato da* $1 + ... + 1 + T(n - n)$

Esempio 7. ***Metodo di sostituzione***
Trattasi della sostituzione della tesi da dimostrare all'argomento della funzione nell'equazione. Supponiamo $T(n) = 2T(n/2) + n; T(n) \in O(nlogn)$*. Nota: la base del logaritmo normalmente è 2, ma non è un'informazione importante per la dimostrazione dell'equazione, è una costante moltiplicativa.*

Dimostriamo la proprietà $T(n) \leq c(nlogn)$ *per induzione; supponiamo che la tesi valga per tutti i numeri più piccoli di* n *e mostriamo che vale per* n*. Come primo passaggio:*

$$T(n) = 2T(n/2) + n; T(n) \in O(nlogn) \implies T(n) \leq 2c(n/2)log(n/2) + n$$

Abbiamo solamente sostituito all'interno dell'argomento della tesi la funzione $T(n)$*. Semplice. Andiamo avanti:*

$$\begin{aligned} T(n) &\leq 2c(n/2)log(n/2) + n \\ &\leq cnlog(n/2) + n \\ &\leq cn(logn - log2) + n \\ &\leq cnlogn - cn + n \\ &\leq cnlogn - (c - 1)n \end{aligned} \tag{1.3}$$

Quindi, quando è che $T(n) \leq cnlogn$*? Bisogna vedere il valore di c sottratto.* $(c - 1) \geq 0 \implies c \geq 1$*.*

Le ipotesi in questi casi sono state supposte dall'esercizio, ma nella costruzione di un algoritmo non le varemo a disposizione; quindi il processo di scelta è basato sulle educated guesses. Si testano le tesi e si vede se funziona, in alternativa si cambia idea e si prova altro.

Se l'algoritmo non è estremamente complesso, è anche possibile utilizzare gli **alberi di ricorrenza**, che mostrano ogni istanza della ricorsione, caso per caso. In questo caso: 1. n 2. n/2 3. n/4 3. n/4 2. n/2 3. n/4 3. n/4 Si andrebbe avanti ad infinitum fino al raggiungimento del caso base. Notare che la somma di tutti i passi risulta 1; ne deduciamo che l'algoritmo si eseguirà n volte.

1.4 Teorema dell'esperto

1.5 Esercizi svolti

Capitolo 2

Ordinamenti e selezioni

2.1 Insertion Sort

2.2 Merge Sort

2.3 Heap Sort

2.4 Quick Sort classico e probabilistico

2.5 Counting Sort

2.6 Radix Sort

2.7 Bucket Sort

2.8 Problema della selezione

Capitolo 3

Strutture dati

3.1 Heap

3.2 Alberi binari

3.2.1 RB-alberi

3.2.2 B-alberi binomiali

3.3 Tabelle hash

3.4 Code con priorità

3.5 Insiemi disgiunti

3.6 Estensione di strutture dati

3.7 Grafi

Capitolo 4

Progetto e analisi di algoritmi

4.1 Divide et impera

4.2 Programmazione greedy

4.3 Programmazione dinamica

4.4 Ricerca locale

4.5 Backtracking

4.6 Branch and bound

Capitolo 5

Algoritmi fondamentali

5.1 Alberi di copertura di costo minimo

Prim, Kruskal

5.2 Programmazione lineare

Simplesso, algoritmo fondamentale polinomiale basato sugli ellissoidi

5.3 Cammini minimi

5.3.1 Sorgente singola

Dijkstra, Bellman-Ford

5.3.2 Sorgente multipla

Floyd-Warshall, Johnson

5.4 Flusso massimo

Ford-Fulkerson, Karp

5.5 Matching massimale su grafo bipartito