2 Notazione Asintotica

Sommario: *In questo capitolo vengono dati gli strumenti formali per calcolare il costo computazionale degli algoritmi. Vengono in particolare definite le notazioni O, Ω e Θ, e viene dato il concetto di pseudocodice, sul quale si mostra come calcolare il costo computazionale dell'algoritmo corrispondente.*

Per poter valutare l'efficienza di un algoritmo, così da poterlo confrontare con algoritmi diversi che risolvono lo stesso problema, bisogna essere in grado di valutarne il suo ***costo computazionale***, ovvero il suo tempo di esecuzione e/o le sue necessità in termini di memoria [1]. In generale, questa valutazione è complicata e contiene spesso dettagli inutili che vorremmo ignorare, per cui ci piacerebbe limitarci a dare una visione più astratta e valutare solo quello che informalmente possiamo chiamare "tasso di crescita", cioè la velocità con cui il tempo di esecuzione cresce all'aumentare della dimensione dell'input. Poiché per piccole dimensioni dell'input il tempo usato è comunque poco, tale valutazione è più interessante quando la dimensione dell'input è sufficientemente grande, per questo si parla di ***efficienza asintotica degli algoritmi***. Inoltre, siccome - come abbiamo già visto - la velocità del-

[1] Salvo ove sia diversamente specificato, la grandezza che viene valutata nel seguito è il tempo di esecuzione dell'algoritmo.

lo specifico calcolatore usato influisce solo per una costante moltiplicativa, vogliamo descrivere un modello che riesca a prescindere da essa.

Quanto detto finora giustifica le seguenti definizioni.

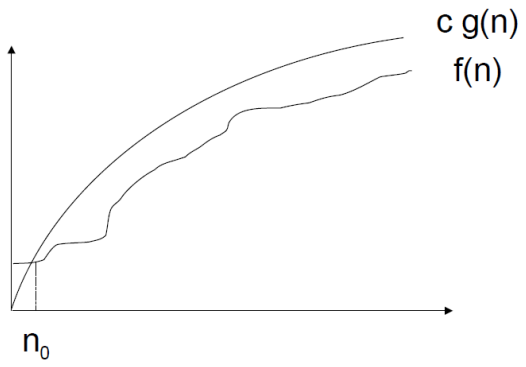
2.1 Notazione O (limite asintotico superiore)

Date due funzioni $f(n)$, $g(n) \geq 0$ si dice che

$$f(n) \text{ è un } O(g(n))$$

se esistono due costanti c ed n_0 tali che

$$0 \leq f(n) \leq c\ g(n) \text{ per ogni } n \geq n_0$$



Esempio 2.1: Sia $f(n) = 3n + 3$.

$f(n)$ è un $O(n^2)$ in quanto, posto $c = 6$, $cn^2 \geq 3n + 3$ per ogni $n \geq 1$.

Ma $f(n)$ è anche un $O(n)$ in quanto $cn \geq 3n + 3$ per ogni $n \geq 1$ se $c \geq 6$, oppure per ogni $n \geq 3$ se $c \geq 4$.

È facile convincersi che, data una funzione $f(n)$, esistono infinite funzioni $g(n)$ per cui $f(n)$ risulta un $O(g(n))$. Come sarà

più chiaro in seguito, ci interessa tentare di determinare la funzione $g(n)$ che meglio approssima la funzione $f(n)$ dall'alto o, informalmente, la più piccola funzione $g(n)$ tale che $f(n)$ sia $O(g(n))$.

Esempio 2.2: Sia $f(n) = n^2 + 4n$.

Ebbene, $f(n)$ è un $O(n^2)$ in quanto $cn^2 \geq n^2 + 4n$ per ogni n se $c \geq 5$, oppure per ogni $n \geq \frac{4}{c-1}$ se $c > 1$.

Esempio 2.3: Sia $f(n)$ un polinomio di grado d: $f(n) = \sum_{i=0}^{d} a_i n^i$, (assumiamo quindi $a_d \neq 0$; se invece $a_d = 0$, allora vorrà dire che il grado di $f(n)$ sarà non d ma minore; le altre costanti, invece, possono essere anche nulle).

Dimostriamo, per induzione su d, che $f(n)$ è un $O(n^d)$.

Caso base:

d = 0: $f(n) = a_0$, quindi $f(n)$ è una costante, cioè un $O(n^0) = O(1)$ per ogni n e per ogni $c \geq a_0$.

Ipotesi induttiva:

per ogni $k<d$, il polinomio di grado k, $h(n) = \sum_{i=0}^{k} a_i n^i$, è un $O(n^k)$. Questo vuol dire che esiste una costante c' tale che $\sum_{i=0}^{k} a_i n^i \leq c' n^k$.

Passo induttivo:

Dobbiamo dimostrare che $f(n) = \sum_{i=0}^{d} a_i n^i$ è un $O(n^d)$, cioè che esiste una costante c tale che $\sum_{i=0}^{d} a_i n^i \leq c\ n^d$.

Si osservi che $f(n)$ si può scrivere come $f(n) = h(n) + a_d n^d$ dove $h(n) = \sum_{i=0}^{k} a_i n^i$ per qualche $k<d$ e che quindi, per l'ipotesi induttiva, esiste una costante c' tale che $h(n) \leq c'\ n^k \leq c'\ n^{d-1}$.

Ora:

$$f(n) = h(n) + a_d n^d \leq c' n^{d-1} + a_d n^d \leq c' n^d + a_d n^d = (c' + a_d) n^d$$

Ponendo $c = c' + a_d$ si ha la tesi.

Alternativamente, possiamo dimostrare la stessa cosa senza usare l'induzione. Sia $c=max\{a_0, a_1, ..., a_d\}$. Allora:

$$f(n) = \sum_{i=0}^{d} a_i n^i \leq \sum_{i=0}^{d} c n^i \leq \sum_{i=0}^{d} c n^d = n^d \sum_{i=0}^{d} c = c(d+1) n^d = c' n^d.$$

2.2 Notazione Ω (limite asintotico inferiore)

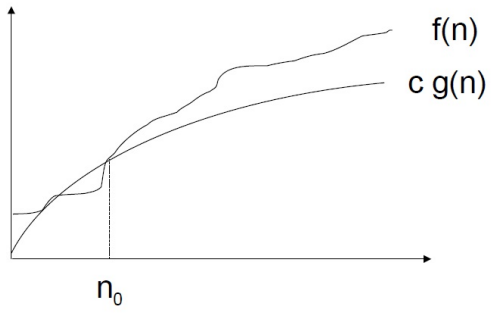
Date due funzioni $f(n)$, $g(n) \geq 0$ si dice che

$$f(n) \text{ è un } \Omega(g(n))$$

se esistono due costanti c ed n_0 tali che

$$f(n) \geq c\ g(n) \ per\ ogni\ n \geq n_0$$

.



Esempio 2.4: Sia $f(n) = 2n^2 + 3$.

$f(n)$ è un $\Omega(n)$ in quanto, posto $c = 1$, $2n^2 + 3 \geq cn$ per qualunque n; ma $f(n)$ è anche un $\Omega(n^2)$ in quanto $2n^2 + 3 \geq cn^2$ per ogni n, se $c \leq 2$.

Esempio 2.5: Sia $f(n)$ un polinomio di grado m, cioè $f(n) = \sum_{i=0}^{m} a_i n^i$, con $a_m > 0$.
La dimostrazione che $f(n)$ è un $\Omega(n^m)$ è analoga alla dimostrazione che $f(n)$ è un $O(n^m)$ e perciò viene lasciata come esercizio.

Abbiamo visto come, in entrambe le notazioni esposte in precedenza, per ogni funzione $f(n)$ sia possibile trovare più funzioni $g(n)$. In effetti $O(g(n))$ e $\Omega(g(n))$ sono ***insiemi di funzioni***, e dire "$f(n)$ è un $O(g(n))$" oppure "$f(n) = O(g(n))$" ha il significato di "$f(n)$ appartiene a $O(g(n))$".
Tuttavia, poiché i limiti asintotici ci servono per stimare con la maggior precisione possibile il costo computazionale di un algoritmo, vorremmo trovare – fra tutte le possibili funzioni $g(n)$ – quella che più si avvicina a $f(n)$.
Per questo cerchiamo la più piccola funzione $g(n)$ per determinare O e la più grande funzione $g(n)$ per determinare Ω.
La definizione che segue formalizza questo concetto intuitivo.

2.3 Notazione Θ (limite asintotico stretto)

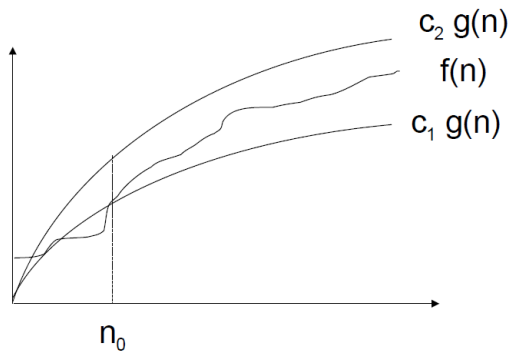
Date due funzioni $f(n)$, $g(n) \geq 0$ si dice che

$$f(n) \text{ è un } \Theta(g(n))$$

se esistono tre costanti c_1, c_2 ed n_0 tali che

$$c_1\ g(n) \leq f(n) \leq c_2\ g(n) \text{ per ogni } n \geq n_0$$

In altre parole, $f(n)$ è $\Theta(g(n))$ se è contemporaneamente $O(g(n))$ e $\Omega(g(n))$.



Esempio 2.6: Sia $f(n) = 3n + 3$.

$f(n)$ è un $\Theta(n)$ ponendo, ad esempio, $c_1 = 3, c_2 = 4, n_0 = 3$.

Esempio 2.7: Dimostrare o confutare che la funzione $(n+10)^3$ è in $\Theta(n^3)$.

- $(n+10)^3$ è $O(n^3)$, infatti: per $n \geq 10$ si ha $(n+10)^3 \leq (2n)^3 = 8n^3$. Quindi basta prendere $n_0 = 10$ e $c = 8$.
- $(n+10)^3$ è $\Omega(n^3)$, infatti: $n^3 < (n+10)^3$. Quindi basta prendere $n_0 = c = 1$.

Segue che $(n+10)^3$ è $\Theta(n^3)$.

Esempio 2.8: Sia $f(n)$ un polinomio di grado m, cioè $f(n) = \sum_{i=0}^{m} a_i n^i$, con $a_m > 0$.
La dimostrazione che $f(n)$ è un $\Theta(n_m)$ discende dagli esempi 2.3 e 2.5.

Si osservi che $\Theta(n^h) \neq \Theta(n^k)$ quando $h \neq k$.

2.4 Algebra sulla notazione asintotica

Per semplificare il calcolo del costo computazionale asintotico degli algoritmi si possono sfruttare delle semplici regole che dapprima enunciamo chiarendole con degli esempi, ed in un secondo momento dimostriamo.

Regole sulle costanti moltiplicative

1A : Per ogni $k > 0$ e per ogni $f(n) \geq 0$, se $f(n)$ è un $O(g(n))$ allora anche $k\ f(n)$ è un $O(g(n))$.

1B : Per ogni $k > 0$ e per ogni $f(n) \geq 0$, se $f(n)$ è un $\Omega(g(n))$ allora anche $k\ f(n)$ è un $\Omega(g(n))$.

1C : Per ogni $k > 0$ e per ogni $f(n) \geq 0$, se $f(n)$ è un $\Theta(g(n))$ allora anche $k\ f(n)$ è un $\Theta g(n))$.

Informalmente, queste tre regole si possono riformulare dicendo che le costanti moltiplicative si possono ignorare.

Regole sulla commutatività con la somma

2A : Per ogni $f(n)$, $d(n) > 0$, se $f(n)$ è un $O(g(n))$ e $d(n)$ è un $O(h(n))$ allora $f(n)+d(n)$ è un $O(g(n)+h(n)) = O(max(g(n),h(n)))$.

2B : Per ogni $f(n)$, $d(n) > 0$, se $f(n)$ è un $\Omega(g(n))$ e $d(n)$ è un $\Omega(h(n))$ allora $f(n)+d(n)$ è un $\Omega(g(n)+h(n)) = \Omega(max(g(n),h(n)))$.

2C : Per ogni $f(n)$, $d(n) > 0$, se $f(n)$ è un $\Theta(g(n))$ e $d(n)$ è un $\Theta(h(n))$ allora $f(n)+d(n)$ è un $\Theta(g(n)+h(n)) = \Theta(max(g(n),h(n)))$.

Informalmente, queste tre regole si possono riformulare dicendo che le notazioni asintotiche commutano con l'operazione di somma.

Regole sulla commutatività col prodotto

3A : Per ogni $f(n)$, $d(n) > 0$, se $f(n)$ è un $O(g(n))$ e $d(n)$ è un $O(h(n))$ allora $f(n)d(n)$ è un $O(g(n)h(n))$.

3B : Per ogni $f(n)$, $d(n) > 0$, se $f(n)$ è un $\Omega(g(n))$ e $d(n)$ è un $\Omega(h(n))$ allora $f(n)d(n)$ è un $\Omega(g(n)h(n))$.

3C : Per ogni $f(n)$, $d(n) > 0$, se $f(n)$ è un $\Theta(g(n))$ e $d(n)$ è un $\Theta(h(n))$ allora $f(n)d(n)$ è un $\Theta(g(n)h(n))$.

Informalmente, queste tre regole si possono riformulare dicendo che le notazioni asintotiche commutano con l'operazione di prodotto.

Prima di dimostrare le regole enunciate, facciamo degli esempi.

Esempio 2.9: Trovare il limite asintotico stretto per $f(n) = 3n2^n + 4n^4$.

$3n2^n + 4n^4 = \Theta(n)\Theta(2^n) + \Theta(n^4) = \Theta(n2^n) + \Theta(n^4) = \Theta(n2^n)$.

Esempio 2.10: Trovare il limite asintotico stretto per $f(n) = 2^{n+1}$.

$2^{n+1} = 2\ 2n = \Theta(2^n)$.

Esempio 2.11: Trovare il limite asintotico stretto per $f(n) = 2^{2n}$.

Dapprima osserviamo che la funzione $2^{2n} = 4^n$ NON è in $O(2^n)$ e la dimostrazione è per assurdo. Assumiamo 4^n in $O(2^n)$ allora esiste una costante c ed un valore n_0 per cui si ha $4^n \leq c2^n$ per $n \geq n_0$, questo significa che, da un certo n in poi, vale $2^n \leq c$ e questo è assurdo perché la funzione 2^n cresce e non può essere limitata da una costante c.

In effetti, $2^{2n} = \Theta(2^{2n})$.

Quest'ultimo esempio ci permette di notare che le ***costanti moltiplicative si possono ignorare solo se non sono all'esponente***.

Esempio 2.12: Trovare il limite asintotico stretto per $f(n) = log_{10}\ n$.

Poiché $log_a\ n = log_a\ b\ log_b\ n$, basta prendere $a = 10$ e $b = 2$, ottenendo che $f(n) = \Theta(log_2\ n)$.

Più in generale, $\Theta(log_a\ n) = \Theta(log_b\ n)$ per ogni coppia di costanti a e b, $a \neq b$.

Dimostriamo ora tutte le regole precedentemente enunciate.

Regola 1A. Per ogni $k > 0$ e per ogni $f(n) \geq 0$, se $f(n)$ è un $O(g(n))$ allora anche $k\ f(n)$ è un $O(g(n))$.

Dim. Per ipotesi, $f(n)$ è un $O(g(n))$ quindi esistono due costanti c ed n_0 tali che:

$$f(n) \leq cg(n) \text{ per ogni } n \geq n_0.$$

Ne segue che:

$$kf(n) \leq kcg(n).$$

Questo prova che, prendendo kc come nuova costante c' e mantenendo lo stesso n_0, $kf(n)$ è un $O(g(n))$. **CVD**

Regola 2A. Per ogni $f(n)$, $d(n) > 0$, se $f(n)$ è un $O(g(n))$ e $d(n)$ è un $O(h(n))$ allora $f(n)+d(n)$ è un $O(g(n)+h(n)) = O(max(g(n),h(n)))$.

Dim. Se $f(n)$ è un $O(g(n))$ e $d(n)$ è un $O(h(n))$ allora esistono quattro costanti c', c'', n'_0 ed n''_0 tali che:

$f(n) \leq c'g(n)$ per ogni $n \geq n'_0$ e $d(n) \leq c''h(n)$ per ogni $n \geq n''_0$

allora:

$$f(n) + d(n) \leq c'g(n) + c''h(n) \leq max(c', c'')(g(n) + h(n))$$

per ogni $n \geq max(n'_0, n''_0)$.

Da ciò segue che $f(n) + d(n)$ è un $O(g(n)+h(n))$.

Infine:

$$max(c', c'')(g(n) + h(n)) \leq 2\ max(c', c'')\ max(g(n), h(n)).$$

Ne segue che $f(n) + d(n)$ è un $O(max(g(n), h(n)))$. **CVD**

Regola 3A. Per ogni $f(n)$, $d(n) > 0$, se $f(n)$ è un $O(g(n))$ e $d(n)$ è un $O(h(n))$ allora $f(n)d(n)$ è un $O(g(n)h(n))$.

Dim. Se $f(n)$ è un $O(g(n))$ e $d(n)$ è un $O(h(n))$ allora esistono quattro costanti c', c'', n'_0 ed n''_0 tali che:

$f(n) \leq c'g(n)$ per ogni $n \geq n'_0$ e $d(n) \leq c''h(n)$ per ogni $n \geq n''_0$

allora:

$$f(n)d(n) \leq c'c''g(n)h(n) \text{ per ogni } n \geq max(n'_0, n''_0).$$

Da ciò segue che $f(n)d(n)$ è un $O(g(n)h(n))$. **CVD**

Le dimostrazioni delle altre regole che coinvolgono le notazioni Ω e Θ sono lasciate per esercizio.

2.5 Calcolo della notazione asintotica tramite limiti

Spesso è possibile determinare la notazione asintotica di una funzione calcolando il limite di un rapporto.

In particolare:

- se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = k > 0$ allora vuol dire che per ogni $\varepsilon > 0$ esiste un n_0 tale che, per ogni n $\geq n_0$, $k - \varepsilon \leq \frac{f(n)}{g(n)} \leq k + \varepsilon$. Essendo $g(n) > 0$ ed $\varepsilon < k$, posto $c_1 = k - \varepsilon$ e $c_2 = k + \varepsilon$, si ha che $c_1\ g(n) \leq f(n) \leq c_2\ g(n)$, cioè $f(n) = \Theta(g(n))$;
- se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ allora $f(n)=\Omega(g(n))$ ma $f(n)\neq\Theta(g(n))$; infatti, per ogni $M>0$, esiste un n_0 tale che, per ogni $n \geq n_0$, $f(n) > M\ g(n)$, mentre non si potrà mai trovare una costante $c > 0$ tale che $f(n) \leq c\ g(n)$;
- con ragionamenti analoghi, se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ allora $f(n)=O(g(n))$ ma $f(n) \neq \Theta(g(n))$.

Ovviamente, quando il limite non esiste, questo metodo non si può usare e bisogna procedere diversamente.

Si lascia come esercizio dimostrare o confutare in due modi diversi (con questo metodo ed usando le definizioni) che:

- 4^n è $O(2^{n \log n})$;
- $(n-50)^2$ è $\Theta(n^2)$.

2.6 Alcune sommatorie notevoli

In questo paragrafo calcoliamo alcune sommatorie notevoli, studiandone il loro andamento asintotico. Sia le tecniche dimostrative che i risultati sono estremamente utili per la risoluzione di esercizi.

Esempio 2.13: Si consideri la sommatoria $S = \sum_{i=1}^{n} i$. Dimostrare che $S = \Theta(n^2)$.

$S = \sum_{i=1}^{n} i = \ldots + \lceil \frac{n}{2} \rceil + \ldots + n \geq \lceil \frac{n}{2} \rceil \lceil \frac{n}{2} \rceil \geq \frac{n^2}{4} = \Omega(n^2)$.

Inoltre, $S = \sum_{i=1}^{n} i = 1 + \ldots + n \leq n + n + \ldots + n = n \cdot n = O(n^2)$

Ne consegue che $S = \Theta(n^2)$.

Più precisamente, possiamo dimostrare che : $S = \frac{n(n+1)}{2}$.

Sommiamo le seguenti due equazioni:

$$\begin{array}{rcccccccccc} S & = & 1 & + & 2 & + & \ldots & + & n-1 & + & n \\ S & = & n & + & n-1 & + & \ldots & + & 2 & + & 1 \\ \hline 2S & = & n+1 & + & n+1 & + & \ldots & + & n+1 & + & n+1 \end{array} = (n+1)n$$

da questo deduciamo che $S = \frac{n(n+1)}{2}$.

Con la stessa tecnica, è possibile far vedere che la sommatoria $S = \sum_{i=1}^{n} i^c$ dove c è una qualsiasi costante intera positiva, è in $S = \Theta(n^{c+1})$. La dimostrazione viene lasciata per esercizio.

Esempio 2.14: Si consideri la sommatoria $S = \sum_{i=0}^{n} 2^i$. Dimostrare che $S = 2^{n+1} - 1$.

Sottraiamo la seconda dalla prima delle due seguenti equazioni:

$$\begin{array}{rcccccccccc} 2S & = & & & 2 & + & 2^2 & + & \dots & + & 2^n & + & 2^{n+1} \\ S & = & 1 & + & 2 & + & \dots & + & 2^{n-1} & + & 2^n & & \\ \hline S & = & -1 & + & 0 & + & \dots & + & 0 & + & 2^{n+1} & = & 2^{n+1} - 1 \end{array}$$

da questo deduciamo che $S = 2^{n+1} - 1$.

Ne segue che $S = \Theta(2^n)$, infatti $2^{n+1} - 1 = 2 \cdot 2^n - 1 = \Theta(2^n)$.

Usando analogo ragionamento, si può far vedere che la sommatoria $S = \sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1}$, dove c è una qualsiasi costante intera positiva diversa da 1; questo vuol dire che $S = \Theta(c^n)$ se $c > 1$ e $S = O(1)$ se $c < 1$. La dimostrazione è lasciata per esercizio.

Esempio 2.15: Si consideri la sommatoria $S = \sum_{i=1}^{n} i2^i$. Dimostrare che $S = (n-1)2^{n+1} + 2$.

Sommiamo le seguenti due equazioni:

$$
\begin{array}{rcccccccccccc}
S & = & 2 & + & 2\cdot 2^2 & + & 3\cdot 2^3 & + & \dots & + & (n-1)\cdot 2^{(n-1)} & + & n2^n \\
2S & = & & & 2^2 & + & 2\cdot 2^3 & + & \dots & + & \dots & + & (n-1)2^n \quad + \quad n2^{n+1} \\
\hline
S-2S & = & 2 & + & 2^2 & + & 2^3 & + & \dots & + & 2^n & - & n2^{n+1}
\end{array}
$$

deduciamo dunque che

$$-S = \sum_{i=1}^{n} 2^i - n2^{n+1} = (2^{n+1}-1) - 1 - n2^{n+1}$$

dove si usa il valore della somma geometrica calcolato nell'esempio precedente. Abbiamo quindi $S = (n-1)2^{n+1} + 2$.
Possiamo anche dimostrare che $\sum_{i=0}^{n} i2^i = \Theta(n2^n)$. Ciò segue dal punto precedente, infatti $(n-1)2^{n+1} - 2 = \Theta(n2^n)$.

Con la stessa tecnica si può valutare la sommatoria $S = \sum_{i=1}^{n} i \cdot c^i$ dove c è una qualsiasi costante maggiore di 1, provando che $S = \frac{nc^{n+1}}{c-1} - \frac{c^{n+1}-1}{(c-1)^2} + 1$ e anche che $S = \Theta(nc^n)$.

Esempio 2.16: Si consideri la sommatoria $S = \sum_{i=1} \log_2 i$.
Dimostrare che $S = \Theta(n \log n)$.
Preliminarmente si osservi che $\sum_{i=1}^{n} \log_2 i = \log_2 \left(\prod_{i=1}^{n} i\right) = \log_2(n!)$ mentre dalla definizione di n! si ha $\left(\frac{n}{2}\right)^{\frac{n}{2}} \leq n! \leq n^n$.

1. $S = \log_2(n!) \leq \log_2 n^n = n \log_2 n$. Quindi $S = O(n \log n)$
2. $S = \log_2(n!) \geq \log_2 \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \log_2 \left(\frac{n}{2}\right) = \frac{n}{2} \log_2 n - \frac{n}{2} =$
   $= \frac{n}{4} \log_2 n + \frac{n}{4} \log_2 n - \frac{n}{2} \geq \frac{n}{4} \log_2 n$
   dove l'ultima diseguaglianza vale per $n \geq n_0 = 4$.
   Quindi $S = \Omega(n \log n)$.

Da 1) e 2) segue $S = \Theta(n \log n)$.

Con analoghi ragionamenti è possibile dimostrare che per la sommatoria $S = \sum_{i=1}^{n} \log_2^c i$ dove c è una qualsiasi costante maggiore di 1, vale che $S = \Theta(n \log^c n)$.

Esempio 2.17: Si consideri la sommatoria $S = \sum_{i=1}^{n} \frac{1}{i}$. Dimostrare che $S = \Theta(\log n)$.

Non è nota una formula chiusa per questa somma (nota come somma armonica). Possiamo però ottenerne stime per difetto e per eccesso ricorrendo agli integrali. Infatti, in generale, quando una somma può essere espressa come $\sum_{i=a}^{b} f(i)$ dove $f(i)$ è una funzione monotona continua non crescente allora possiamo approssimarla tramite integrali:

$$\int_a^{b+1} f(x)dx \leq \sum_{i=a}^{b} f(i) \leq \int_{a-1}^{b} f(x)dx.$$

Applicando il metodo al nostro caso abbiamo:

1. $\sum_{i=1}^{n} \frac{1}{i} \geq \int_1^{n+1} \frac{1}{x} dx = [\ln x]_1^{n+1} = \ln(n+1) - \ln 1 = \ln(n+1)$.
   Quindi $S = \Omega(\log n)$.

2. $\sum_{i=1}^{n} \frac{1}{i} = 1 + \sum_{i=2}^{n} \ln x \leq 1 + \int_1^n \frac{1}{x} dx = 1 + [\ln x]_1^n = 1 + \ln n - \ln 1 = 1 + \ln n$.
   Quindi $S = O(\log n)$.

Da 1) e 2) deduciamo $S = \Theta(\log n)$. Più precisamente abbiamo:

$$\ln(n+1) < S < \ln n + 1.$$

Utilizzando le limitazioni offerte dagli integrali:

$$\int_{a-1}^{b} f(x)dx \leq \sum_{i=a}^{b} f(i) \leq \int_{a}^{b+1} f(x)dx$$

con $f(x)$ continua e crescente lasciamo per esercizio dimostrare che:

- $\sum_{i=1}^{n} i^c = \Theta(n^{c+1})$.
- $\sum_{i=1}^{n} c^i = \Theta(c^n)$.

2.7 Valutazione del costo di un algoritmo

Vediamo ora come calcolare effettivamente il costo computazionale di un algoritmo, adottando il criterio della misura di costo uniforme descritto nel par. 1.4.2.

Prima di procedere, facciamo due importanti considerazioni.

Innanzi tutto, osserviamo che è ragionevole pensare che il costo computazionale, inteso come funzione che rappresenta il tempo di esecuzione di un algoritmo, sia una funzione monotona non decrescente della dimensione dell'input. Questa osservazione ci conduce a constatare che, prima di passare al calcolo del costo, bisogna definire quale sia la **dimensione dell'input**. Trovare questo parametro è, di solito, abbastanza semplice: in un algoritmo di ordinamento esso sarà il numero di elementi da ordinare, in un algoritmo che lavora su una matrice sarà il numero di righe e di colonne, in un algoritmo che

opera su alberi sarà il numero di nodi, eccetera. Tuttavia, vi sono casi in cui l'individuazione del parametro non è banale; in ogni caso, è necessario stabilire quale sia la variabile (o le variabili) di riferimento *prima* di accingersi a calcolare il costo.

In secondo luogo, vogliamo sottolineare che la notazione asintotica viene sfruttata pesantemente per il calcolo del costo computazionale degli algoritmi, quindi - in base alla definizione stessa - tale costo computazionale potrà essere ritenuto valido *solo* asintoticamente.

In effetti, esistono degli algoritmi che per dimensioni dell'input relativamente piccole hanno un certo comportamento, mentre per dimensioni maggiori un altro.

Infine, a volte il costo di un algoritmo non dipende solo dalla dimensione dell'input ma anche dal valore esatto dell'input; in questi casi è necessario suddividere lo studio del costo computazionale in termini di **caso migliore** e **caso peggiore**. A parità di dimensione dell'input, il caso peggiore si riferisce a dei dati per i quali l'algoritmo deve eseguire il massimo numero di operazioni; viceversa, il caso migliore corrisponde al minor numero di operazioni.

Per poter valutare il tempo computazionale di un algoritmo, esso deve essere formulato in un modo che sia chiaro, sintetico e non ambiguo.

Si adotta il cosiddetto ***pseudocodice***, che è una sorta di linguaggio di programmazione "informale" nell'ambito del quale:

- si impiegano, come nei linguaggi di programmazione, i costrutti di controllo (for, if then else, while, ecc.);
- si impiega il linguaggio naturale per specificare le operazioni;
- si ignorano i problemi di ingegneria del software;
- si omette la gestione degli errori, al fine di esprimere solo l'essenza della soluzione.

Non esiste una notazione universalmente accettata per lo pseudocodice. In queste dispense useremo ove possibile, il linguaggio Python, che è sufficientemente vicino al linguaggio naturale (quindi, ad esempio, il simbolo = per indicare un'assegnazione, il simbolo == per verificare che il contenuto di 2 variabili sia lo stesso, il simbolo != per verificare che il contenuto di 2 variabili sia differente, l'indentazione per rappresentare i diversi livelli dei blocchi di istruzioni, ecc.). Al fine di concentrare l'attenzione sull'essenza degli algoritmi, nello pseudocodice si evita la gestione degli errori dell'input. Inoltre, nonostante la libertà lasciata da questo strumento, è consigliato utilizzare dei nomi evocativi per le variabili, in modo da massimizzare la leggibilità.

Le regole generali che si adottano per valutare il tempo computazionale di un algoritmo sono le seguenti:

- le **istruzioni elementari** (operazioni aritmetiche, lettura del valore di una variabile, assegnazione di un valore a una variabile, valutazione di una condizione logica su un numero costante di operandi, stampa del valore di una variabile, ecc.) hanno costo $\Theta(1)$;
- l'istruzione if (condizione): istruz1 else: istruz2 ha costo pari al costo di verifica della condizione (di solito costante) più il massimo dei costi di istruz1 e istruz2;
- le istruzioni iterative (o iterazioni, come ad esempio i cicli for e while) hanno un costo pari alla somma dei costi massimi di ciascuna delle iterazioni (compreso il costo di verifica della condizione). Se tutte le iterazioni hanno lo stesso costo massimo, allora il costo dell'iterazione è pari al prodotto del costo massimo di una singola iterazione per il numero di iterazioni; in entrambi i casi è comunque necessario stimare il numero delle iterazioni. Si osservi che la condizione viene valutata una volta in più rispetto al numero delle iterazioni, poiché l'ultima valutazione, che dà esito negativo, è quella che fa terminare l'iterazione;
- il costo dell'algoritmo è pari alla somma dei costi delle istruzioni che lo compongono.

Esempio 2.18: Calcolo del massimo in un array A disordinato contenente n valori (e che si assume non vuoto).

```
def Trova_Max(A):
1   n=len(A)                    Θ(1)
2   max = A[0]                  Θ(1)
3   for i in range(1,n):        (n - 1) iterazioni più Θ(1)
4      if A[i] > max:           Θ(1)
5         max = A[i]            Θ(1)
6   return max                  Θ(1)
```

Il costo delle istruzioni 1 e 2 è $\Theta(1)$. L'iterazione viene eseguita $(n - 1)$ volte, e ciascuna iterazione (costituita dalle istruzioni 3, 4 e 5) ha costo $\Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$ poiché l'incremento del contatore (istr. 3), la valutazione della condizione (istr. 4) e l'assegnazione (istr. 5) sono istruzioni elementari.
Il costo dell'istruzione 6 è $\Theta(1)$.
Quindi, detto $T(n)$ il costo computazionale di questo algoritmo, esso vale:

$$T(n) = \Theta(1) + [(n - 1)\ \Theta(1) + \Theta(1)] + \Theta(1) = \Theta(n).$$

Vogliamo ora mostrare come variano i tempi di esecuzione di un algoritmo in funzione del suo costo computazionale.
Ipotizziamo di disporre di un sistema di calcolo in grado di effettuare una operazione elementare in un nanosecondo (10^9 operazioni al secondo), e supponiamo che la dimensione del problema sia $n = 10^6$ (un milione):

- costo $O(n)$ - tempo di esecuzione: 1 millesimo di secondo;
- costo $O(n \log n)$ - tempo di esecuzione: 20 millesimi di secondo;
- costo $O(n^2)$ - tempo di esecuzione: 1000 secondi = 16 minuti e 40 secondi.

C'è un'altra situazione interessante da considerare: che succede se il costo computazionale cresce esponenzialmente, ad esempio quando è $O(2^n)$?
È abbastanza ovvio che i tempi di esecuzione diventano rapidamente proibitivi: un tale tipo di problema su un input di dimensione $n = 100$ richiede per la sua soluzione mediante il sistema di calcolo di cui sopra ben $1{,}26*10^{21}$ secondi, cioè circa $3*10^{13}$ anni.
Si potrebbe ipotizzare che l'avanzamento tecnologico, magari formidabile, possa prima o poi rendere abbordabile un tale problema, ma purtroppo non è così.
Infatti, poniamoci la seguente domanda: supponendo di avere un calcolatore estremamente potente che riesce a risolvere un problema di dimensione $n = 1000$, avente costo computazionale $O(2^n)$, in un determinato tempo T, quale dimensione $n' = n + x$ del problema riusciremmo a risolvere nello stesso tempo utilizzando un calcolatore mille volte più veloce?
Possiamo scrivere la seguente uguaglianza:

$$T = \frac{2^{1000}\ \textit{operazioni}}{10^k\ \textit{operazioni al secondo}} = \frac{2^{1000+x}\ \textit{operazioni}}{10^{k+3}\ \textit{operazioni al secondo}}$$

Si ha quindi:

$$\frac{2^{1000+x}}{2^{1000}} = 2^x = \frac{10^{k+3}}{10^k} = 10^3 = 1000$$

ossia

$$x = log\ 1000 \approx 10.$$

Dunque, con un calcolatore mille volte più veloce riusciremmo solo a risolvere, nello stesso tempo, un problema di dimensione 1010 anziché di dimensione 1000.

In effetti esiste un'importantissima branca della teoria della complessità che si occupa proprio di caratterizzare i cosiddetti problemi

textbfintrattabili, ossia quei problemi il cui costo computazionale è tale per cui essi non sono né saranno mai risolubili per dimensioni realistiche dell'input.

Ma concentriamoci su problemi decisamente più semplici e cerchiamo di capire se, nel nostro piccolo, riusciamo a perseguire l'**efficienza**; infatti, non è importante solo risolvere un problema, ma risolverlo in modo efficiente, progettando cioè un algoritmo che, tra tutti quelli che risolvono il problema, abbia costo computazionale minore possibile.

Esempio 2.19: Calcolo della somma dei primi n interi.

```
def Trova_Somma_1(n):
    somma = 0                    Θ(1)
    for i in range(1,n):         n iterazioni più Θ(1)
        somma += 1               Θ(1)
```

```
4   return somma                    Θ(1)
```

Il costo dell'istruzione 1 è $\Theta(1)$.

L'iterazione viene eseguita n volte, e ciascuna iterazione (costituita dalle istruzioni 2 e 3) ha costo $\Theta(1) + \Theta(1) = \Theta(1)$. Il costo dell'istruzione 4 è $\Theta(1)$. Quindi, detto $T(n)$ il costo computazionale di questo algoritmo, esso vale:

$$T(n) = \Theta(1) + [n\ \Theta(1) + \Theta(1)] + \Theta(1) = \Theta(n)$$

Osserviamo, tuttavia, che lo stesso problema può essere risolto in modo ben più efficiente come segue:

```
def Calcola_Somma_2(n: intero):
1   somma = n * (n + 1) / 2     Θ(1)
2   return somma                    Θ(1)
```

Il costo della funzione è, ovviamente, $\Theta(1)$, costo decisamente migliore rispetto al $\Theta(n)$ precedente.

Esempio 2.20: Valutazione del polinomio $\sum_{i=0}^{n} a_i x^i$ nel punto $x = c$.

Supponendo che i coefficienti $a_0, a_1, \ldots, a_n$ siano memorizzati nell'array A:

```
def Calcola_Polinomio_1(A, c):
1   somma = A[0]                        Θ(1)
2   for i in range(len(A)):             n iterazioni più Θ(1)
3      potenza = 1                      Θ(1)
4      for j in range(i):               n iterazioni più Θ(1)
```

```
5        potenza=c*potenza              Θ(1)
6     somma=somma + A[i]*potenza        Θ(1)
7  return somma                         Θ(1)
```

Il costo computazionale dell'istruzione 1 è $\Theta(1)$.

La prima iterazione (istruzione 2) viene eseguita n volte; ciascuna iterazione contiene le 4 istruzioni 3-6 il cui costo è globalmente $\Theta(i)$ poiché vi è un ciclo eseguito i volte con, all'interno, operazioni costanti.

Il costo computazionale dell'istruzione 7 è, infine, $\Theta(1)$.

Quindi, detto $T(n)$ il costo computazionale di questo algoritmo, esso vale:

$$T(n) = \Theta(1) + \left[\sum_{i=1}^{n} (\Theta(1)+\Theta(i)) + \Theta(1) \right] + \Theta(1) = \Theta(1)+\Theta(n)+\Theta(n^2) = \Theta(n^2)$$

Osserviamo tuttavia che, solo riscrivendo in modo più oculato lo pseudocodice dello stesso algoritmo, il medesimo problema può essere risolto in modo più efficiente come segue:

```
def Calcola_Polinomio_2(A, c):
1  somma = A[0]                          Θ(1)
2  potenza = 1                           Θ(1)
3  for i in range(1, len(A)):            n iterazioni più Θ(1)
4     potenza=c*potenza                  Θ(1)
5     somma=somma + A[i]*potenza         Θ(1)
6  return somma                          Θ(1)
```

In questa versione della funzione abbiamo semplicemente evitato di ricalcolare più volte le potenze di c, sfruttando quelle

calcolate precedentemente. Il costo computazionale di questa funzione è, ovviamente, $\Theta(n)$, costo decisamente migliore rispetto al $\Theta(n^2)$ precedente.

Concludiamo questo capitolo con un'osservazione.
Un dato algoritmo potrebbe avere tempi di esecuzione (e quindi costo computazionale) diversi a seconda dello specifico input, e non solo della sua dimensione. In tal caso, per fare uno studio esauriente del suo tempo computazionale, bisogna valutare prima quali siano i cosiddetti casi migliore e peggiore, cioè cercare di capire quale input sia particolarmente vantaggioso, ai fini del costo computazionale dell'algoritmo, e quale invece svantaggioso.
Per avere un'idea di quale sia il tempo di esecuzione atteso di un algoritmo, a prescindere dall'input, è chiaro che è necessario prendere in considerazione il caso peggiore, cioè la situazione che porta alla computazione più onerosa. Nel contempo, però, vorremmo essere il più precisi possibile e quindi, nel contesto del caso peggiore, cerchiamo di calcolare il costo in termini di notazione asintotica Θ. Laddove questo non sia possibile, essa dovrà essere approssimata sia per difetto (tramite la notazione Ω) che per eccesso (tramite la notazione O).

3 Il problema della ricerca

Sommario: *In questo capitolo si affronta un problema basilare, quello della ricerca, e vengono proposte due soluzioni classiche: la ricerca sequenziale (nel caso in cui i dati non siano ordinati) e la ricerca binaria o dicotomica (nel caso in cui i dati siano ordinati).*

Nell'informatica esistono alcuni problemi particolarmente rilevanti, poiché essi:

- si incontrano in una grande varietà di situazioni reali;
- appaiono come sottoproblemi da risolvere nell'ambito di problemi più complessi.

Uno di questi problemi è la ricerca di un elemento in un insieme di dati (ad es. numeri, cognomi, ecc.).
Iniziamo a definire un po' più formalmente tale problema descrivendone l'input e l'output:

- **Input**: un array A di n valori (interi, stringhe, ecc.) ed un valore v;
- **Output**: un indice i tale che $A[i] = v$, oppure -1 se il valore v non è presente nell'array.