2) Notazione asintotica

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 valutare l'ordine di grandezza del suo ***costo computazionale***, ovvero del suo tempo di esecuzione e delle sue necessità in termini di memoria. In particolare, tale valutazione ha senso quando la dimensione dell'input è sufficientemente grande.

Per questo si parla di ***efficienza asintotica degli algoritmi***.

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

Prima di entrare nel vivo della valutazione del costo computazionale asintotico degli algoritmi, è necessario introdurre alcune definizioni.

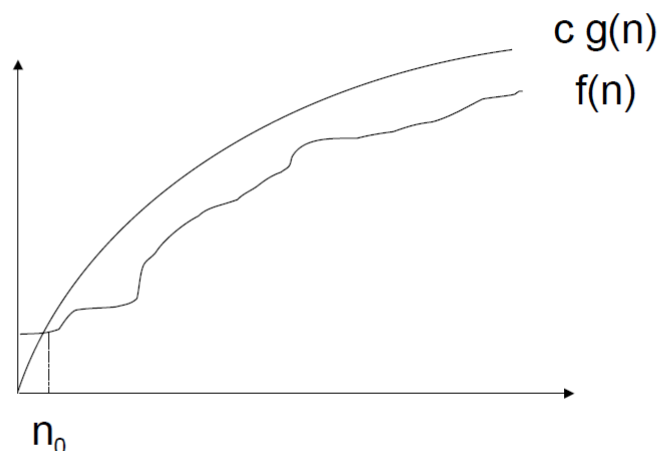
2.1 Notazione *O* (limite asintotico superiore)

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 $c \geq 4$.

E' 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 $n \geq 4/(c-1)$ se $c > 1$.

Esempio 2.3

Sia *f(n)* un polinomio di grado *m*: $f(n) = \sum_{i=0}^{m} a_i n^i$, con $a_m > 0$

Dimostriamo, per induzione su *m*, che *f(n)* è un $O(n^m)$ aggiungendo anche che $c \geq \sum_{i=0}^{m} a_i$.

Casi base

- ***m* = 0**: $f(n) = a_0$, quindi *f(n)* è una costante, cioè un *O(1)* per ogni *n* e per ogni $c \geq a_0$
- ***m* = 1**: $f(n) = a_0 + a_1 n$, quindi *f(n)* è un *O(n)* per qualunque *n*, per ogni $c \geq a_0 + a_1$.

Ipotesi induttiva

$h(n) = \sum_{i=0}^{m-1} a_i n^i$ è un $O(n^{m-1})$ se $c \geq \sum_{i=0}^{m-1} a_i$

Passo induttivo

Dobbiamo dimostrare che

$$f(n) = \sum_{i=0}^{m} a_i n^i \text{ è un } O(n^m)$$

cioè che

$$c' n^m \geq \sum_{i=0}^{m} a_i n^i \text{ con } c' \geq \sum_{i=0}^{m} a_i$$

Si osservi che $f(n)$ si può scrivere come $f(n) = h(n) + a_m n^m$ e che, per l'ipotesi induttiva:

- $cn^{m-1} \geq h(n)$ con $c \geq \sum_{i=0}^{m-1} a_i$

Ora:

$$f(n) = h(n) + a_m n^m \leq cn^{m-1} + a_m n^m \leq cn^m + a_m n^m = (c + a_m) n^m$$

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

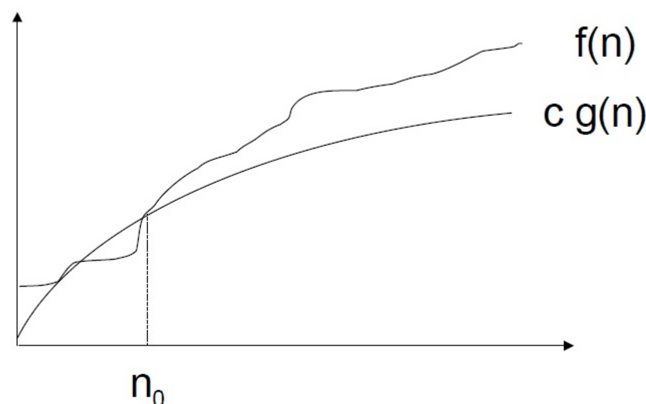
2.2 Notazione Ω (limite asintotico inferiore)

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

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



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*: $f(n) = \sum_{i=0}^{m} a_i n^i$, 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 *Ω(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)

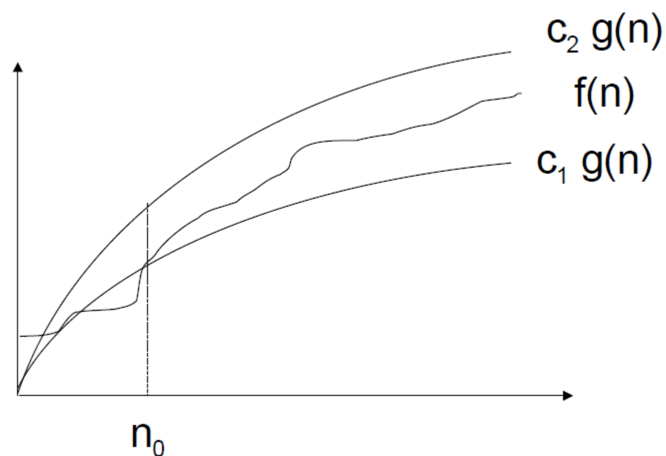
Date due funzioni $f(n), g(n) \geq 0$ si dice che

f(n) è un *Θ(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 *Θ(g(n))* se è contemporaneamente *O(g(n))* e *Ω(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

Sia $f(n)$ un polinomio di grado m: $f(n) = \sum_{i=0}^{m} a_i n^i$, con $a_m > 0$

La dimostrazione che $f(n)$ è un $\Theta(n^m)$ discende dagli esempi 2.3 e 2.5.

2.4 Algebra della 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 sopra, facciamo alcuni esempi.

Esempio 2.8

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 2.9

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

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

Esempio 2.10

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

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

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

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' e c'', n'_0 ed n''_0 tali che:

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

allora:

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

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

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' e c'', n'_0 ed n''_0 tali che:

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

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 Valutazione del costo computazionale di un algoritmo

Vediamo ora come calcolare effettivamente il costo computazionale di un algoritmo, adottiando 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 dati, 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, ecc.. 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.

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 (come del resto nel libo di testo) useremo l'indentazione per rappresentare i diversi livelli dei blocchi di istruzioni, useremo il simbolo ← per indicare un'assegnazione, il simbolo = per verificare che il contenuto di 2 variabili sia lo stesso e il simbolo ≠ per verificare che il contenuto di 2 variabili sia differente.

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) then istruzione1 else istruzione2` ha costo pari al costo di verifica della condizione (di solito costante) più il massimo dei costi di `istruzione1` e `istruzione2`;
- le istruzioni iterative (o iterazioni: cicli `for`, `while` e `repeat`) 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.

Un dato algoritmo potrebbe avere tempi di esecuzione (e quindi costo computazionale) diversi a seconda dell'input. 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 per difetto (tramite la notazione Ω) e per eccesso (tramite la notazione O).

Esempio 2.11

Calcolo del massimo in un vettore disordinato contenente *n* valori.

```
Funzione Trova_Max (A: vettore)
1     max ← A[1]                          Θ(1)
2     for i = 2 to n do                   (n – 1) iterazioni più Θ(1)
3           if A[i] > max                 Θ(1)
4                 then max ← A[i]         Θ(1)
5     stampa max                          Θ(1)
```

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

L'iterazione viene eseguita $(n - 1)$ volte, e ciascuna iterazione (costituita dalle istruzioni 2, 3 e 4) ha costo $\Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$ poiché l'incremento del contatore (istr. 2), la valutazione della condizione (istr. 3) e l'assegnazione (istr. 4) sono istruzioni elementari.

Il costo dell'istruzione 5 è $\Theta(1)$.

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

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

Esempio 2.12

Calcolo della somma dei primi *n* interi.

```
Funzione Calcola_Somma(n: intero)
1     somma ← 0                           Θ(1)
2     for i = 1 to n do                   n iterazioni più Θ(1)
3           aggiungi i a somma            Θ(1)
4     stampa 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:

```
Funzione Calcola_Somma(n: intero)
1       somma ← n (n-1)/2                Θ(1)
2       stampa somma                     Θ(1)
```

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

Approfittiamo di questo esempio per sottolineare che l'**efficienza** in un algoritmo va sempre perseguita, e che 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 2.13

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

Supponendo che i coefficienti $a_0, a_1, \ldots, a_{n-1}$ siano memorizzati nel vettore A:

```
Funzione Calcola_Polinomio(n: intero, A: vettore, c reale)
1     somma ← A[0]                                    Θ(1)
2     for i = 1 to n do                               n iterazioni più Θ(1)
3          potenza ← 1                                Θ(1)
4          for j = 1 to i do                          i iterazioni più Θ(1)
5               potenza ← c*potenza                   Θ(1)
6               somma ← somma + A[i]*potenza          Θ(1)
7     stampa 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:

```
Funzione Calcola_Polinomio(n: intero, A: vettore, c reale)
1     somma ← A[0]                                    Θ(1)
3          potenza ← 1                                Θ(1)
2     for i = 1 to n do                               n iterazioni più Θ(1)
5               potenza ← c*potenza                   Θ(1)
6               somma ← somma + A[i]*potenza          Θ(1)
7     stampa 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.

Ribadiamo che, nel progettare un algoritmo, l'efficienza deve sempre essere perseguita in modo prioritario.

Concludiamo questo argomento mostrando 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;
- comsto $O(n^2)$ - 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 $O(2^n)$?

E' 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 $1{,}26 \cdot 10^{21}$ secondi, cioè circa $3 \cdot 10^{13}$ 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:

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

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