Notazione asintotica per costo computazionale

Come faccio a dire se un algoritmo è più efficiente di un altro (per risolvere uno stesso problema).
Per definire il costo computazionale ci basta una stima, questa stima è data da:

- Tempo: # operazioni nel caso peggiore
- Spazio

A partire da un certo n io dovrò sapere esattamente cosa farà il mio algoritmo
La notazione asintotica è comoda per descrivere la funzione tempo di esecuzione nel caso peggiore; a volte però possiamo estendere la notazione al dominio dei numeri reali o limitarla ad un sottoinsieme dei naturali.
Utilizzeremo la n.a. soprattutto per descrivere i tempi di esecuzione degli algoritmi ma può essere applicata anche a funzioni che descrivono qualche altro aspetto degli algoritmi.

Stima:

- **Upper Bound** (limite superiore): non capiterà che l'algoritmo abbia bisogno di più di questo numero di operazioni → $O(\cdot)$
- **Lower Bound** (limite inferiore): # minimo di operazioni per ogni nel caso peggiore → $\Omega(\cdot)$
- **Tight Bound**: U.B. e L.B. si equivalgono → $\ominus(\cdot)$

La prima cosa da fare è cercare un U.B. più preciso possibile, di questa funzione ci interessa l'ordine di grandezza.

Date 2 funzioni:

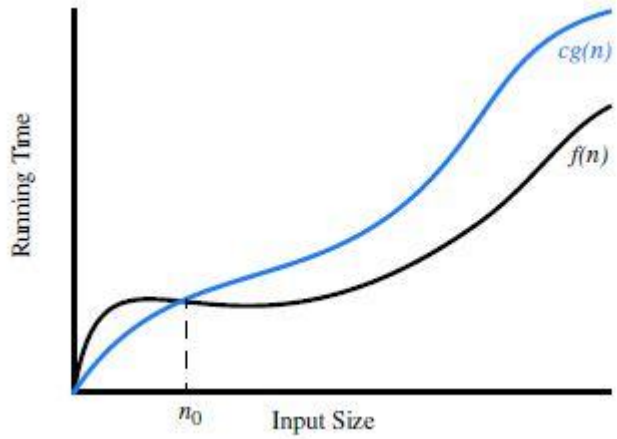
$f(n): N \rightarrow \mathbb{R}$
$g(n): N \rightarrow \mathbb{R}$

Diremo che f(n) ∈ O(g(n))

Se e solo se

∃ 2 valori: $c > 0$, $n_0 > 0$ tali che:

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



ESEMPIO
$stima\ del\ tempo \rightarrow T(n) \leq f(n) = 2n^2 + 3n + 6$
$g(n) = n^2$

$f(n) \in O(n^2)\ nel\ caso\ peggiore\ il\ tempo\ si\ comporta\ circa\ come\ una\ parabola$

$2n^2 + 3n + 6 \leq 2n^2 + 3n^2 + 6 \leq\ 2n^2 + 3n^2 + n^2 = 6n^2 = cn^2$

f(n) si comporterà come una parabola.
Ci concentriamo sul termine di ordine più grande ignorando le costanti, ci interessa il **comportamento.**

Esempi:

1. Ciclo semplice:

```
for i = 0 to n - 1
    c := c + 1
    d := d + c
c := c + d
```

All'interno del ciclo for sono presenti 2 operazioni, il ciclo for esegue n iterazioni, quindi:
$\#op. = 2 \cdot iterazioni$

> $T(n) = 2n + 1 \in O(n)$

2. Ciclo annidato

```
for i = 0 to n - 1
    for j = i to n - 1
        x := x + 1
    y := y + 1
```

> $T(n) = n + \frac{n(n+1)}{2} \in O(n^2)$

$n \rightarrow$ `y := y+ 1`

i	j	# iterazioni ciclo interno
0	0 - n-1	n
1	1 ... n-1	n-1
2	2 ... n-1	n-2
k	k... n-1	n-k
n - 1	n-1 ... n-1	1

$\hookrightarrow n + (n-1) + (n-2) + ... + 2 + 1$
Sto sommando i primi n numeri naturali

= $\frac{n(n+1)}{2}$

Upper Bound

Diremo che:

$$f(n) \in O(g(n)) \text{ se esistono delle costanti positive c e } n_0 : 0 \leq f(n) \leq c \cdot g(n)\ \forall n \geq n_0$$

Notiamo che la notazione $f(n) \in \Theta(g(n))$ implica $f(n) = O(g(n))$, in quanto la notazione Θ è più forte della notazione O.
Qualsiasi funzione quadratica $an^2 + bn + c$, con $a > 0$, è in $\Theta(n^2)$ implica anche che tali funzioni quadratiche siano tutte in $O(n^2)$.

Espressione O()	Nome
O(1)	Costante
O($loglogn$)	loglog

Espressione O()	Nome
O($\sqrt[c]{n}$, c > 1)	sublineare
O(n)	lineare
O(nlogn)	nlogn
O(n^2)	quadratica
O(n^3)	cubica
O(n^k), k > 1	polinomiale
O(a^n), a > 1	esponenziale
O($n!$)	fattoriale

Più siamo in alto nella tabella più il nostro algoritmo sarà efficente.

Esempio

> Vediamo un esempio. $Sia f(n) = 2n^2 + 3n + 6, g(n) = n^2$, e tentiamo di mostrare che $2n^2 + 3n + 6 = O(n^2)$.
> Occorrerà quindi provare che esistono costanti c > 0 e $n_0 \in \mathbb{N}$ per cui $2n^2 + 3n + 6 \leq cn^2$, per ogni n ≥ n0.
> A tal fine osserviamo che $2n^2 + 3n + 6 \leq 2n^2 + 3n^2 + 6 \leq 2n2 + 3n^2 + n^2 = 6n^2$, per ogni valore di n per cui $n^2 \geq 6$, ovvero per ogni valore di n ≥ 3.
> Quindi, le costanti c e n_0 che cercavamo per provare che $2n^2 + 3n + 6 \leq cn^2$, per ogni $n \geq n_0$, possono essere scelte come c = 6 e n_0 = 3. Ovviamente, possono esistere anche altri valori di c e n_0 per cui la diseguaglianza $2n^2 + 3n + 6 \leq cn^2$, per ogni $n \geq n_0$, sia soddisfatta. Tuttavia, al fine di provare che $2n^2 + 3n + 6 = O(n^2)$ ci basta aver provato che $c = 2$ e $n_0 = 3$ vanno bene.

L'esempio descritto ammette una semplice generalizzazione, ovvero per ogni k costante

Generico polinomio di grado k:

$f(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n^1 + a_0 n^0 \in O(n^k)$

Questa proprietà vale per tutti i polinomi
La difficoltà sta nel definire f(n)

$0 \leq i \leq k$

$a_i n^i \leq |a_i| n^i \leq |a_i| n^k \dots \leq |a_k| n^k + |a_{k-1}| n^k + \dots + |a_1| n^k + |a_0| n^k = (|a_k| + |a_{k-1}| + \dots + |a_0|) n^k = cn^k$

Alcune proprietà:

1. Se $f(n) \in O(g(n)) allora : a \cdot f(n) \in O(g(n)) \ \forall a > 0$ costante
   Esempio:
   $7logn \in O(n)$
2. $$Se \begin{cases} f(n) \in O(g(n)), \\ d(n) \in O(h(n)) \end{cases} \rightarrow \begin{aligned} f(n) + d(n) &\in O(g(n) + h(n)) \\ &\in O(max(g(n), h(n))) \end{aligned}$$
3. $$Se \begin{cases} f(n) \in O(g(n)), \\ d(n) \in O(h(n)) \end{cases} \rightarrow f(n) * d(n) \in O(g(n) * h(n))$$

4.

$$Se \begin{cases} f(n) \in O(g(n)), \\ g(n) \in O(h(n)) \end{cases} \rightarrow f(n) \in O(h(n))$$

Lower Bound

Diremo che $f(n) \in \Omega(g(n))$ se e solo se $\exists c > 0, n_0 \geq 0$
per cui $f(n) \geq c \cdot g(n) \forall n \geq n_0$

Espressione Ω()	**Nome**
Ω(1)	Costante
$\Omega(loglogn)$	loglog
$\Omega(\sqrt[c]{n}$, c > 1)	sublineare
Ω(n)	lineare
Ω(nlogn)	nlogn
$\Omega(n^2)$	quadratica
$\Omega(n^3)$	cubica
$\Omega(n^k)$, k > 1	polinomiale
$\Omega(a^n)$, a > 1	esponenziale
$\Omega(n!)$	fattoriale

L'ordine consta ancora ma stavolta è il contrario di prima.

Tight Bound

Diremo che $f(n) \in \Theta(g(n))$
se e solo se:

$\exists c1, c2 > 0, n_0 \geq 0$ per cui
$\forall n \geq n_0 \quad c1 \cdot g(n) \leq f(n) \leq c2 \cdot g(n)$

La definizione di $\Theta(g(n))$ richiede che ogni membro $f(n) \in \Theta(g(n))$ sia ***asintoticamente non negativo***, ovvero che $f(n)$ sia non negativa quando n è sufficentemente grande.

> *Stessa tabella di prima ma con θ.*

Problema

Un problema è trattabile se ha come U.B. un polinomiale
ovvero: se $\exists$ un algrotimo A di tempo polinomiale

$$T(n) \in O(n^k)$$

Un problema è *intrattabile* se $\nexists$ un A polinomiale per risolverlo.
Un problema potrebbe anche essere non risolvibile.
Esiste anche una classe di problemi per cui non si è dimostrato se esiste o non esiste una soluzione, essi sono detti NP-compatibili.

Esempi

1. Input:
    - Sequenza di numeri interi

      ```
      A[0]...A[n-1]
      ```

    - $x \in N$

   Output: la posizione di c in A (se presente), altrimenti "non c'è"

   ```
   Cerca(A, x)
       i:= 0
       while i < n and A[i] != x
           i:=i+1

       if i=n
           then return "non c'è"
           else return i
   ```

2. Stesso problema ma stavolta la sequenza è ordinata in ordine non decrescente

   ```
   A[0] <= A[1] <= ... <= A[n-1]
   ```

   Output: posizione di x in A

   Ricerca binaria (Dicotomica)

   Quand'è che non lo trovo? Quando la sequenza rimane vuota senza che io lo abbia trovato

   Pseudo codice:

   ```
   Bin_Search(A, n, x)
       i:=0, j:=n-1
       while i <= J
           k = int_floor((i+j)/2)
           if A[k] = x then return k
           if A[k] > x then j:=k-1
                       else then i:=k+1
       return "non c'è"
   ```

   Quindi, ricapitolando:
   Se A non è ordinato $\rightarrow$ O(n)
   Se A è ordinato $\rightarrow$ O(logn)

	Algoritmo	**Problema**
U.B.	Riesce a risolvere con $T_a(n) \in O(\cdot)$	Si può risolvere con $T_p(n) \in O(\cdot)$
L.B.	Non può fare meglio di $T_a(n) \in \Omega(\cdot)$	Non posso risolvere il problema $T_p(n) \in \Omega(\cdot)$