Complessità di un algoritmo

Si può calcolare in base a quante risorse usa un determinato algoritmo:

- **tempo** ovvero quanto tempo ci mette
- **spazio** ovvero quanta memoria occorre per eseguire l'algoritmo
- **hardware** ovvero numero di processori etc..

In relazione al tempo, ci interessa stabilire la **complessità temporale**:

- Ciò è utile per capire quanto ci vuole ad eseguire un determinato algoritmo
- E' inoltre utile per confrontare l'efficienza di diversi algoritmi che risolvono lo stesso problema
- Anche per stimare la grandezza massima dell'**input**
- Migliorare l'algoritmo analizzando le parti di codice che vengono eseguite più volte

Cosa si intende per **tempo** però?
Possiamo dare diverse definizioni:

- **numero di secondi** (dipende dalla macchina)
- **numero di operazioni elementari**
- **numero di** volte che viene eseguita **una specifica operazione**.

Minimo di un vettore

Ad esempio, per la ricerca dell'**elemento minimo in un vettore**, la complessità temporale è **lineare**.
Tuttavia come varia il **tempo** in funzione della dimensione

dell'**ingresso**?
Per dimensione si intende:
$|m| =$dimensione di m $=$ num. bit per rappresentare m $=$ parte intera di $\log_2(m) + 1$.

La dimensione dell'**input** dunque conta.
Non vale però la seguente relazione: $|x| = |y| \rightarrow T(x) = T(y)$.

Ad esempio: se devo ordinare due vettori lunghi entrambi ***n*** *non è detto che ci metterò lo stesso tempo per entrambi*

Dunque:

$T_{migliore}(n) = min\{T(X) : |x| = n\}$
Caso migliore: $T_{peggiore}(|x|) = T(x)$

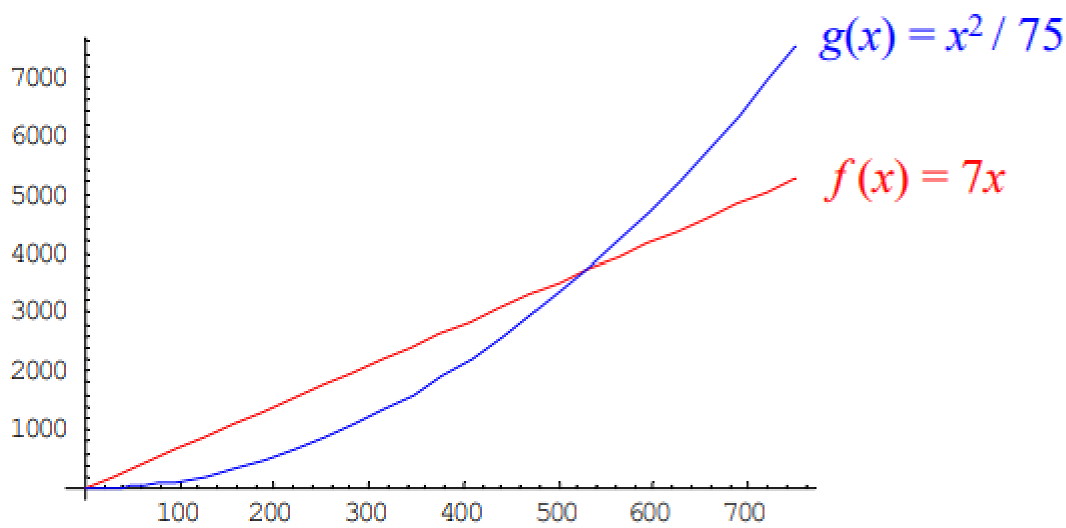
Minimo(A,j,k): quando il minimo è in $A[j]$
Insert-Sort:: vettore non decrescente

$T_{peggiore}(n) = max\{T(X) : |x| = n\}$
Caso migliore: $T_{peggiore}(|x|) = T(x)$

Minimo(A,j,k): quando $A[j..k]$ è ordinato in senso decrescente
Insert-Sort:: vettore decrescente

Come confrontare le funzioni

Visto che il **tempo di calcolo** non è un numero ma si esprime come una funzione, per confrontare due algoritmi dobbiamo dunque confrontare le due funzioni associate

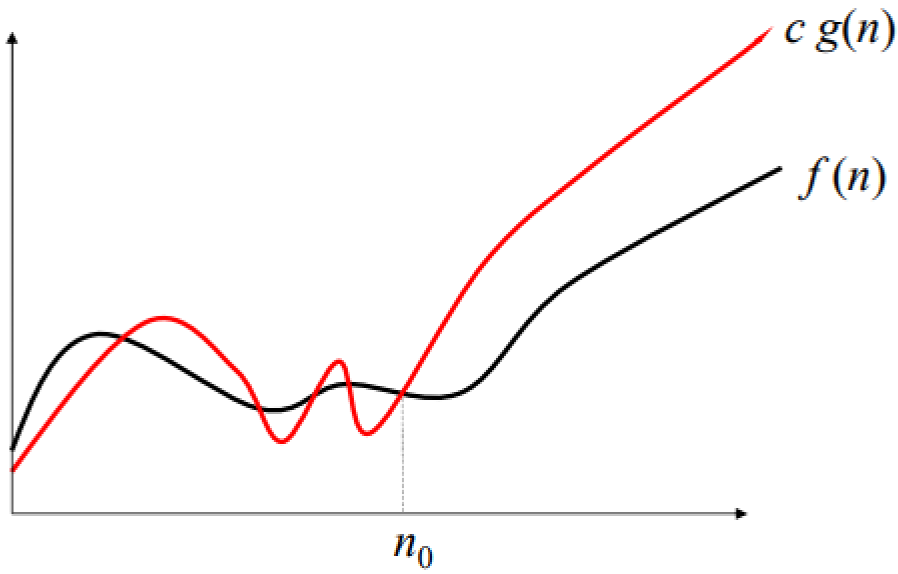


Ad esempio, fino a $x = 525$, $g(x) < f(x)$.
Tuttavia se trascuriamo quel numero finito di casi, $g(x) > f(x)$
Si parla dunque di **complessità asintotica** se ignoriamo quel numero finito di casi.
Inoltre le **costanti** contano poco:

- Infatti se una funzione cresce come n e un'altra funzione come $2n$, entrambe crescono **linearmente**, dunque **asintoticamente**(con numeri molto grandi) questa costante moltiplicativa è trascurabile
- Moltiplicando per una costante il tempo di calcolo, la massima dimensione trattabile cambia di poco
- La stima esatta delle costanti è difficile

Ordini di grandezza: O-grande

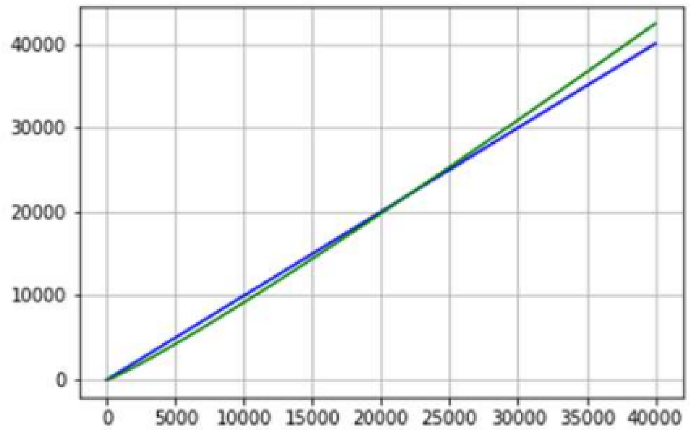
Matematicamente, una funziona $f(x)$ è O di un'altra funzione $g(x)$(in notazione $f(x) \in O(g(x))$) se $f(x)$ cresce **al più** come $g(x)$.
Detto in parole povere la velocità di crescita di $f(x)$ è $\leq$ di $g(x)$
Esempio: $f(x) = 3x \in O(g(x) = x)$ *per quanto possa essere controintuitivo(ripassate analisi bestie)*



$$f(n) \in O(g(n)) \Leftrightarrow \exists c > 0, n_0 \forall n > n_0 . f(n) \leq cg(n)$$

Definizione matematica esatta di O-grande

Perchè ciò è importante?
Visto che il tempo di calcolo è espresso sotto forma di una funzione, ci interessa capire come cresce con numeri molto grandi.
Dunque se un algoritmo impiega $T(x) = 3x$ e un algoritmo $T'(x) = x$, sostanzialmente il loro tempo computazionale è uguale visto che $T(x) \in O(T'(x))$



- $f(n) = \frac{n}{10} \cdot \log(n+2), g(n) = n$
- si vede anche graficamente ma bisogna plottare per grandi valori di n
- (non a caso $n \cdot \log n$ si chiama anche "quasi lineare")

Esempio di funzione NON O-grande di f(n)=n

Vari O-grande

Le funzioni che hanno un tempo computazionale costante, sono O(1).

Esempio: L'inserimento di un elemento in una lista è una operazione con tempo computazionale costante, infatti ciò non dipende dal numero degli elementi.
*Dunque **Inserimento di una lista** $\in O(1)$*

Inoltre, se $p(n)$ è un polinomio di grado k, allora $p(n) \in O(n^k)$
Esempio: $3n^3 + 2n^2 \in O(n^3)$, infatti il polinomio alla sinistra è di grado 3, dunque è $O(n^3)$

Per quanto riguarda i **logaritmi** invece:
$\log_a n = \frac{\log_b n}{\log_b a} = \frac{1}{\log_b a} \log_b n$
Dal momento che $\frac{1}{\log_b a}$ è una costante, applicando la definizione,

otteniamo che $O(\log_a n) = O(\log_b n)$
e scriviamo semplicemente
$O(\log n)$

Le **esponenziali** invece crescono sempre più velocemente di quelle polinomiali.
Esempio: $O(n^2) \subset O(2^n)$

Inoltre $O(2^n) \neq O(3^n)$, infatti $O(2^n) \in O(3^n)$

Velocità di crescita:
$O(1) \subset O(\log n) \subset \sqrt{n} \subset n \subset n \log n \subset n^2 \subset n^3 \subset 2^n \subset 3^n \ldots$

Confini stretti

L'O-grande descrive solamente il comportamento di una funzione rispetto ad un'altra per limiti asintotici **superiori**

Tuttavia se vogliamo esprimere il comportamento di una funzione per un limite **inferiore**, dobbiamo usare altre notazioni

Omega e Theta

Una funzione è $f(x) \in \Omega(g(x))$ (in parole: f è Omega di $g)se\$f(x)$
cresce ALMENO come $g(x)$
Detto in parole povere, la velocità di $f \geq g$
(La velocità di f è $\geq$ di g, non il loro valore!)

Una funzione, inoltre, è $f(x) \in \Theta(g(x))$ se $c_1 g(x) \leq f(x) \leq c_2 g(x)$
In parole povere, $f(x)$ cresce come $g(x)$

Dunque:
$f(x) \in \Theta(g(x)) \leftrightarrow f(x) \in \Omega(g(x)) \wedge f(x) \in O(g(x)$
(f è Θ di g **se e solo se** f è O di g e f è Ω di g)

Algebra(informale) degli O-grande

$$f(n) = O(f(n))$$
$$c \cdot O(f(n)) = O(f(n)) \qquad c \text{ costante}$$
$$O(f(n)) + O(g(n)) = O(f(n) + g(n))$$
$$O(f(n)) + O(f(n)) = O(f(n))$$
$$f(n) \in O(g(n)) \Rightarrow O(f(n)) + O(g(n)) = O(g(n))$$
$$O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$$

Esempio:

- $f(n) = 1$
  $O(f(n) + f(n)) = O(1 + 1) = O(2) = O(1)$
- $f(n) = n \; g(n) = n^2$
  $O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(n^2 + n) = O(n^2)$
  $$f(n) \in O(g(n)) \Rightarrow O(f(n)) + O(g(n)) = O(g(n))$$
  $f(n) \in O(g(n))$ *visto che* n *cresce* $\leq$ *di* n^2