

# Tecniche di analisi degli algoritmi

Moreno Marzolla  
marzolla@cs.unibo.it

Dipartimento di Scienze dell'Informazione, Università di Bologna

14 ottobre 2011

Copyright ©2009, 2010 Moreno Marzolla, Università di Bologna

This work is licensed under the Creative Commons Attribution-ShareAlike License. To view a copy of this license, visit

<http://creativecommons.org/licenses/by-sa/3.0/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

# Modello di calcolo

Consideriamo un modello di calcolo costituito da una **macchina a registri** così composta:

- Esiste un dispositivo di input e un dispositivo di output;
- La macchina ha  $N$  locazioni di memoria, con indirizzo da 1 a  $N$ ; ciascuna locazione può contenere un valore (intero, reale...);
- l'accesso in lettura o scrittura ad una qualsiasi locazione richiede **tempo costante**;
- La macchina dispone di un set di registri per mantenere i parametri necessari alle operazioni elementari e per il puntatore all'istruzione corrente;
- La macchina ha un programma composto da un insieme **finito** di istruzioni

# Costo computazionale

## Definizione

*Indichiamo con  $f(n)$  la quantità di **risorse** (tempo di esecuzione, oppure occupazione di memoria) richiesta da un algoritmo su input di dimensione  $n$ , operante su una macchina a registri.*

Siamo interessati a studiare l'*ordine di grandezza* di  $f(n)$  ignorando le costanti moltiplicative e termini di ordine inferiore.

# Misura del costo computazionale

Utilizzare il tempo effettivo di esecuzione di un programma come costo computazionale presenta numerosi svantaggi:

- Implementare un dato algoritmo può essere laborioso;
- Il tempo è legato alla specifica implementazione (linguaggio di programmazione usato, caratteristiche della macchina usata per effettuare le misure, ...);
- Potremmo essere interessati a stimare il costo computazionale usando input troppo grandi per le caratteristiche della macchina su cui effettuiamo le misure;
- Determinare l'ordine di grandezza a partire da misure empiriche non è sempre possibile;

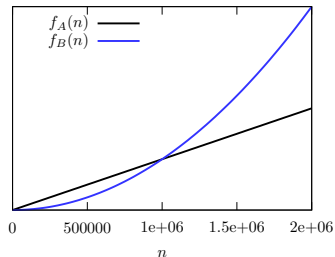
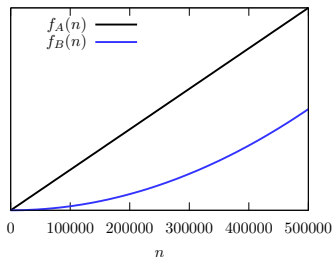
# Costo computazionale

## Esempio

Consideriamo due algoritmi  $A$  e  $B$  che risolvono lo stesso problema.

- Sia  $f_A(n) = 10^3 n$  il costo computazionale di  $A$ ;
- Sia  $f_B(n) = 10^{-3} n^2$  il costo computazionale di  $B$ .

Quale dei due è preferibile?



# La notazione asintotica $O(f(n))$

## Definizione

Data una funzione costo  $f(n)$ , definiamo l'insieme  $O(f(n))$  come l'insieme delle funzioni  $g(n)$  per le quali esistono costanti  $c > 0$  e  $n_0 \geq 0$  per cui vale:

$$\forall n \geq n_0 : g(n) \leq cf(n)$$

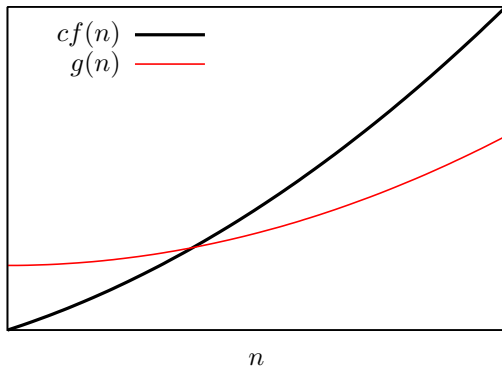
In maniera piú sintetica:

$$O(f(n)) = \{g(n) : \exists c > 0, n_0 \geq 0 \text{ tali che } \forall n \geq n_0 : g(n) \leq cf(n)\}$$

Nota: si utilizza la notazione (sebbene non formalmente corretta)  $g(n) = O(f(n))$  per indicare  $g(n) \in O(f(n))$ .

# Rappresentazione grafica

$$g(n) = O(f(n))$$





# Esempio

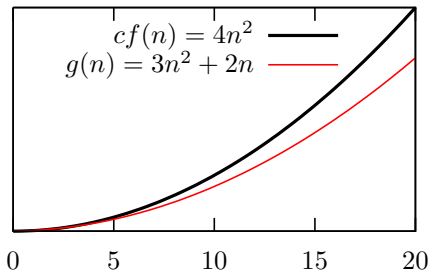
Sia  $g(n) = 3n^2 + 2n$  e  $f(n) = n^2$ . Dimostriamo che  $g(n) = O(f(n))$ .

Dobbiamo trovare due costanti  $c > 0, n_0 \geq 0$  tali che  $g(n) \leq cf(n)$  per ogni  $n \geq n_0$ , ossia:

$$3n^2 + 2n \leq cn^2 \quad (1)$$

$$c \geq \frac{3n^2 + 2n}{n^2} = 3 + \frac{2}{n}$$

se ad esempio scegliamo  $n_0 = 10$  e  $c = 4$ , si ha che la relazione (1) è verificata.



# La notazione asintotica $\Omega(f(n))$

## Definizione

Data una funzione costo  $f(n)$ , definiamo l'insieme  $\Omega(f(n))$  come l'insieme delle funzioni  $g(n)$  per le quali esistono costanti  $c > 0$  e  $n_0 \geq 0$  per cui vale:

$$\forall n \geq n_0 : g(n) \geq cf(n)$$

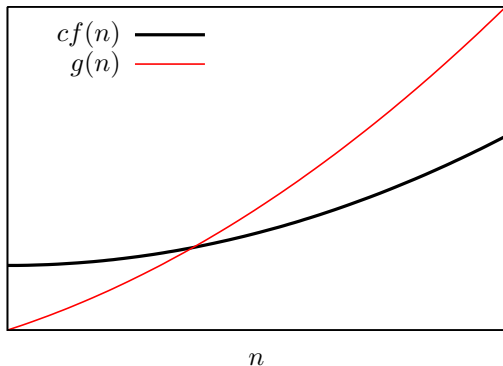
In maniera piú sintetica:

$$\Omega(f(n)) = \{g(n) : \exists c > 0, n_0 \geq 0 \text{ tali che } \forall n \geq n_0 : g(n) \geq cf(n)\}$$

Nota: si utilizza la notazione  $g(n) = \Omega(f(n))$  per indicare  $g(n) \in \Omega(f(n))$ .

# Rappresentazione grafica

$$g(n) = \Omega(f(n))$$





# La notazione asintotica $\Theta(f(n))$

## Definizione

Data una funzione costo  $f(n)$ , definiamo l'insieme  $\Theta(f(n))$  come l'insieme delle funzioni  $g(n)$  per le quali esistono costanti  $c_1 > 0$ ,  $c_2 > 0$  e  $n_0 \geq 0$  per cui vale:

$$\forall n \geq n_0 : c_1 f(n) \leq g(n) \leq c_2 f(n)$$

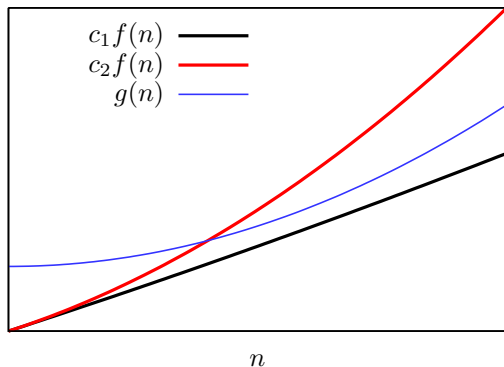
In maniera piú sintetica:

$$\Theta(f(n)) = \{g(n) : \exists c_1 > 0, c_2 > 0, n_0 \geq 0 \text{ tali che} \\ \forall n \geq n_0 : c_1 f(n) \leq g(n) \leq c_2 f(n)\}$$

Nota: si utilizza la notazione  $g(n) = \Theta(f(n))$  per indicare  $g(n) \in \Theta(f(n))$ .

# Rappresentazione grafica

$$g(n) = \Theta(f(n))$$



# Spiegazione intuitiva

- Se  $g(n) = O(f(n))$  significa che l'ordine di grandezza di  $g(n)$  è “minore o uguale” a quello di  $f(n)$ ;
- Se  $g(n) = \Theta(f(n))$  significa che  $g(n)$  e  $f(n)$  hanno lo stesso ordine di grandezza;
- Se  $g(n) = \Omega(f(n))$  significa che l'ordine di grandezza di  $g(n)$  è “maggiore o uguale” a quello di  $f(n)$ .

# Alcune proprietà delle notazioni asintotica

## Simmetria

$g(n) = \Theta(f(n))$  se e solo se  $f(n) = \Theta(g(n))$

## Simmetria Trasposta

$g(n) = O(f(n))$  se e solo se  $f(n) = \Omega(g(n))$

## Transitività

Se  $g(n) = O(f(n))$  e  $f(n) = O(h(n))$ , allora  $g(n) = O(h(n))$ .  
Lo stesso vale per  $\Omega$  e  $\Theta$ .



# Ordini di grandezza

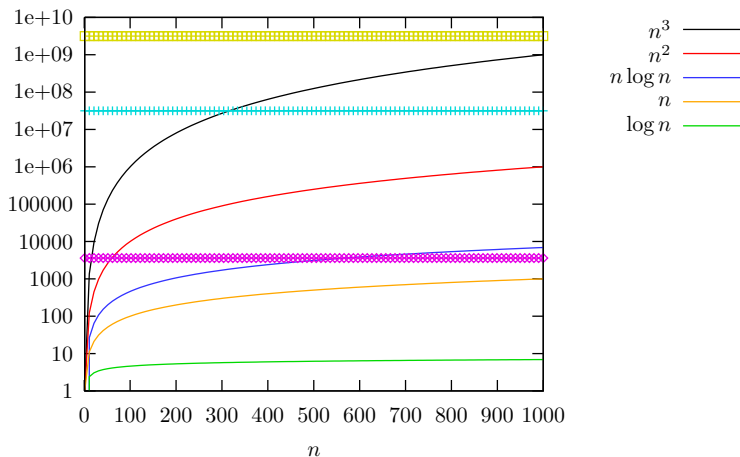
In ordine di costo crescente:

	Ordine	Esempio
$O(1)$	costante	Determinare se un numero è pari
$O(\log n)$	logaritmico	Ricerca di un elemento in un array ordinato
$O(n)$	lineare	Ricerca di un elemento in un array disordinato
$O(n \log n)$	pseudolineare	Ordinamento mediante Merge Sort
$O(n^2)$	quadratico	Ordinamento mediante Bubble Sort
$O(n^3)$	cubico	Prodotto di due matrix $n \times n$ con l'algoritmo "intuitivo"
$O(c^n)$	esponenziale, base $c > 1$	Calcolare il determinante di una matrice mediante espansione dei minori
$O(n!)$	fattoriale	
$O(n^n)$	esponenziale, base $n$	

In generale:

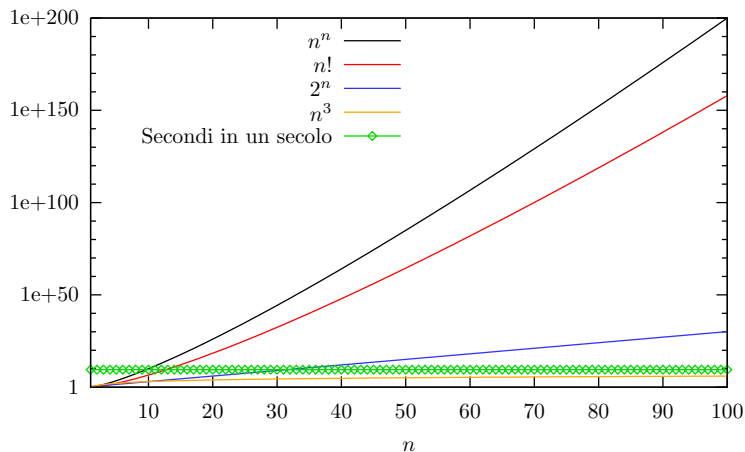
- $O(n^k)$  con  $k > 0$  è **ordine polinomiale**
- $O(c^n)$  con  $c > 1$  è **ordine esponenziale**

# Confronto grafico tra gli ordini di grandezza



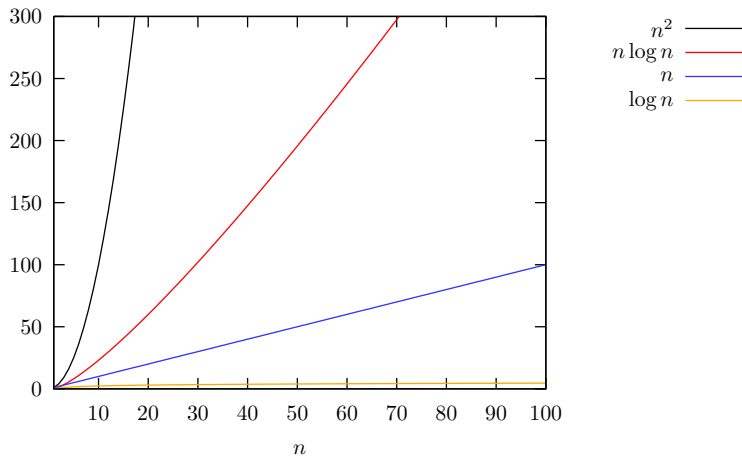
Nota: scala y logaritmica; le linee orizzontali segnano il numero di secondi in un'ora, in un anno e in un secolo (rispettivamente, dal basso verso l'alto)

# Confronto grafico tra gli ordini di grandezza



Nota: scala y logaritmica!

# Confronto grafico tra gli ordini di grandezza



# Vero o falso?

$$6n^2 = \Omega(n^3) ?$$

Applicando la definizione, dobbiamo dimostrare se

$$\exists c > 0, n_0 \geq 0 : \forall n \geq n_0 \quad 6n^2 \geq cn^3$$

Cioè  $c \leq 6/n$ .

Fissato  $c$  è sempre possibile scegliere un valore di  $n$  sufficientemente grande tale che  $6/n < c$ , per cui l'affermazione è **falsa**.  $\square$

# Vero o falso?

$$10n^3 + 2n^2 + 7 = O(n^3) ?$$

Applicando la definizione, dobbiamo dimostrare se

$$\exists c > 0, n_0 \geq 0 : \forall n \geq n_0 \quad 10n^3 + 2n^2 + 7 \leq cn^3$$

Possiamo scrivere:

$$\begin{aligned} 10n^3 + 2n^2 + 7 &\leq 10n^3 + 2n^3 + 7n^3 && (\text{se } n \geq 1) \\ &= 19n^3 \end{aligned}$$

Quindi la disuguaglianza è verificata ponendo  $n_0 = 1$  e  $c = 19$ . □

# Domande

- Dimostrare che  $\log_2 n = O(n)$ ;
- Cosa cambia se il logaritmo di cui sopra non è in base 2?
- Dimostrare che  $n \log n = O(n^2)$ ;
- Dimostrare che, per ogni  $\alpha > 0$ ,  $\log n = O(n^\alpha)$  (suggerimento: da quanto visto sopra si può affermare che  $\log n^\alpha = O(n^\alpha)$ , quindi...)
- Dove collochereste  $O(\sqrt{n})$  nella tabella degli ordini di grandezza? Perché?

# Costo di esecuzione

## Definizione

Un algoritmo  $\mathcal{A}$  ha **costo di esecuzione**  $O(f(n))$  su istanze di ingresso di dimensione  $n$  rispetto ad una certa **risorsa di calcolo** se la quantità  $r(n)$  di risorsa sufficiente per eseguire  $\mathcal{A}$  su una qualunque istanza di dimensione  $n$  verifica la relazione  $r(n) = O(f(n))$ .

**Nota** Risorsa di calcolo per noi significa **tempo di esecuzione** oppure **occupazione di memoria**.



# Complessità dei problemi

## Definizione

*Un problema  $\mathcal{P}$  ha **complessità**  $O(f(n))$  rispetto ad una data risorsa di calcolo se esiste un algoritmo che risolve  $\mathcal{P}$  il cui costo di esecuzione rispetto a quella risorsa è  $O(f(n))$ .*

# Alcune regole utili

## Somma

Se  $g_1(n) = O(f_1(n))$  e  $g_2(n) = O(f_2(n))$ , allora  
 $g_1(n) + g_2(n) = O(f_1(n) + f_2(n))$

## Prodotto

Se  $g_1(n) = O(f_1(n))$  e  $g_2(n) = O(f_2(n))$ , allora  
 $g_1(n) \cdot g_2(n) = O(f_1(n) \cdot f_2(n))$

## Eliminazione costanti

Se  $g(n) = O(f(n))$ , allora  $a \cdot g(n) = O(f(n))$  per ogni costante  $a > 0$

# Osservazione

Utilizzando gli ordini di grandezza, ogni operazione elementare ha costo  $O(1)$ ; un contributo diverso viene dalle istruzioni **condizionali** e **iterative**.

```
if ( F_test ) {  
    F_true  
} else {  
    F_false  
}
```

Supponendo:

- $F_{\text{test}} = O(f(n))$
- $F_{\text{true}} = O(g(n))$
- $F_{\text{false}} = O(h(n))$

Allora il costo di esecuzione del blocco if-then-else è

$$O(\max\{f(n), g(n), h(n)\})$$

# Analisi nel caso ottimo, pessimo e medio

Sia  $\mathcal{I}_n$  l'insieme di tutte le possibili *istanze di input* di lunghezza  $n$ . Sia  $T(I)$  il tempo di esecuzione dell'algoritmo sull'istanza  $I \in \mathcal{I}_n$ .

- Il costo nel **caso pessimo** (*worst case*) è definito come

$$T_{\text{worst}}(n) = \max_{I \in \mathcal{I}_n} T(I)$$

- Il costo nel **caso ottimo** (*best case*) è definito come

$$T_{\text{best}}(n) = \min_{I \in \mathcal{I}_n} T(I)$$

- Il costo nel **caso medio** (*average case*) è definita come

$$T_{\text{avg}}(n) = \sum_{I \in \mathcal{I}_n} T(I)P(I)$$

dove  $P(I)$  è la probabilità che l'istanza  $I$  si presenti.

# Analisi di algoritmi non ricorsivi

Ricerca il valore minimo contenuto in un array non vuoto

```
// Restituisce la posizione dell'elemento minimo in
algoritmo Minimo( A[1..n] di float ) -> int
  int m:=1; // Posizione dell'elemento minimo
  for i:=2 to n do
    if ( A[i]<A[m] ) then
      m = i;
    endif
  endfor
  return m;
}
```

## Analisi

- Sia  $n$  la lunghezza del vettore  $v$ .
- Il corpo del ciclo viene eseguito  $n - 1$  volte;
- Ogni iterazione ha costo  $O(1)$
- Il costo di esecuzione della funzione `Minimo` rispetto al tempo è quindi  $O(n)$  (o meglio,  $\Theta(n)$ : **perché?**).

# Ricerca sequenziale

## Caso ottimo e pessimo

```
// Restituisce la posizione della prima occorrenza del
// nell'array A[1..n]. Ritorna -1 se il valore non e'
algoritmo Trova( array A[1..n] di int, int val ) -> int
    for i:=1 to n do
        if ( A[i]==val ) then
            return i;
        endif
    endfor
    return -1;
```

- Nel **caso ottimo** l'elemento è all'inizio della lista, e viene trovato alla prima iterazione. Quindi  $T_{\text{best}}(n) = O(1)$
- Nel **caso pessimo** l'elemento non è presente nella lista (oppure è presente nell'ultima posizione), quindi si itera su tutti gli elementi. Quindi  $T_{\text{worst}}(n) = \Theta(n)$
- ...E nel **caso medio**?

# Ricerca sequenziale

Analisi del caso medio

Non avendo informazioni sulla probabilità con cui si presentano i valori nella lista, dobbiamo fare delle ipotesi semplificative.

Assumiamo che, dato un vettore di  $n$  elementi, la probabilità  $P_i$  che l'elemento cercato si trovi in posizione  $i$  ( $i = 1, 2, \dots, n$ ) sia  $P_i = 1/n$ , per ogni  $i$  (assumiamo che l'elemento sia sempre presente).

Il tempo  $T(i)$  necessario per individuare l'elemento nella posizione  $i$ -esima è  $T(i) = i$ .

Quindi possiamo concludere che:

$$T_{\text{avg}}(n) = \sum_{i=1}^n P_i T(i) = \frac{1}{n} \sum_{i=1}^n i = \frac{1}{n} \frac{n(n-1)}{2} = \Theta(n)$$

# Esempio

Un algoritmo iterativo di ordinamento

```
public class SortingAlgo {  
    // Calcola l'indice dell'elemento di valore  
    // minimo nell'insieme v[i], v[i+1]... v[j]  
    static int Min( int v[], int i, int j )  
    { /* ... */  
    // v[] deve essere non vuoto  
    public static void Sort( int v[] )  
    {  
        for ( int i=0; i<v.length-1; ++i ) {  
            int m = SortingAlgo.Min( v, i, v.length-1 );  
            // Scambia v[i] e v[m]  
            int tmp = v[i];  
            v[i] = v[m];  
            v[m] = tmp;  
        }  
    }  
}
```



# Analisi dell'algoritmo di ordinamento

- La chiamata  $\text{Min}(v, i, v.\text{length}-1)$  individua l'elemento minimo nell'array  $v[i], v[i+1], \dots, v[n-1]$ . Il tempo richiesto è proporzionale a  $n-i, i=0, 1, \dots, n-1$  (perché?);
- L'operazione di scambio ha costo  $O(1)$  in termini di tempo di esecuzione;
- Il corpo del ciclo `for` viene eseguito  $n$  volte.

Il costo di esecuzione rispetto al tempo dell'intera funzione `Sort` è:

$$\sum_{i=0}^{n-1} (n-i) = n^2 - \sum_{i=0}^{n-1} i = n^2 - \frac{n(n-1)}{2} = \frac{n^2 + n}{2}$$

che è  $\Theta(n^2)$ .

# Analisi di algoritmi ricorsivi

Ricerca di un elemento in un array ordinato

```
public class RicercaBinaria {  
  
    static int TrovaRic( int val, int v[], int i, int j ) {  
        if ( i>j ) { return -1; }  
        else {  
            int m = (i+j)/2;  
            if ( v[m] == val ) { return m; } // trovato  
            else {  
                if ( v[m] > val ) {  
                    return TrovaRic( val, v, i, m-1 );  
                } else {  
                    return TrovaRic( val, v, m+1, j );  
                }  
            }  
        }  
    }  
}  
  
// Trova la posizione di un elemento di valore val nel  
// vettore v[], che deve essere ordinato in senso crescente  
public static int Trova( int val, int v[] ) {  
    RicercaBinaria.TrovaRic( val, v, 0, v.length-1 );  
}  
}
```

# Analisi dell'algoritmo di ricerca binaria

Sia  $T(n)$  il tempo di esecuzione della funzione `TrovaRic` su un vettore di  $n = j - i + 1$  elementi.

In generale  $T(n)$  dipende non solo dal numero di elementi su cui fare la ricerca, ma anche dalla posizione dell'elemento cercato (oppure dal fatto che l'elemento non sia presente).

- Nell'ipotesi più favorevole (**caso ottimo**) l'elemento cercato è proprio quello che occupa posizione centrale; in tal caso  $T(n) = O(1)$ .
- Nel caso meno favorevole (**caso pessimo**) l'elemento cercato non esiste. Quanto vale  $T(n)$  in tale situazione?

# Analisi dell'algoritmo di ricerca binaria

Metodo dell'iterazione

Possiamo definire  $T(n)$  per ricorrenza, come segue.

$$T(n) = \begin{cases} c_1 & \text{se } n = 0 \\ T(\lfloor n/2 \rfloor) + c_2 & \text{se } n > 0 \end{cases}$$

Il **metodo dell'iterazione** consiste nello sviluppare l'equazione di ricorrenza, per intuirne la soluzione:

$$T(n) = T(n/2) + c_2 = T(n/4) + 2c_2 = T(n/8) + 3c_2 = \dots = T(n/2^i) + i \times c_2$$

Supponendo che  $n$  sia una potenza di 2, ci fermiamo quando  $n/2^i = 1$ , ossia  $i = \log n$ . Alla fine abbiamo

$$T(n) = c_1 + c_2 \log n = O(\log n)$$

# Verificare equazioni di ricorrenza

Metodo della sostituzione

Consiste nell'applicare il principio di induzione per verificare la soluzione di una equazione di ricorrenza.

**Esempio** Dimostrare che  $T(n) = O(n)$  è soluzione di

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ T(\lfloor n/2 \rfloor) + n & \text{se } n > 1 \end{cases}$$

**Dimostrazione** Per induzione, verifichiamo che  $T(n) \leq cn$  per  $n$  sufficientemente grande.

- Caso base:  $T(1) = 1 \leq c \times 1$ . Basta scegliere  $c \geq 1$ .
- Induzione:

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + n \\ &\leq c\lfloor n/2 \rfloor + n \quad (\text{ipotesi induttiva}) \\ &\leq cn/2 + n = f(c)n \end{aligned}$$

con  $f(c) = (c/2 + 1)$ . La dimostrazione del passo induttivo funziona quando  $f(c) \leq c$ , ossia  $c \geq 2$ .

# Teorema fondamentale della ricorrenza

## Master Theorem

### Teorema

La relazione di ricorrenza:

$$T(n) = \begin{cases} aT(n/b) + f(n) & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases} \quad (3)$$

ha soluzione:

- 1  $T(n) = \Theta(n^{\log_b a})$  se  $f(n) = O(n^{\log_b a - \epsilon})$  per  $\epsilon > 0$ ;
- 2  $T(n) = \Theta(n^{\log_b a} \log n)$  se  $f(n) = \Theta(n^{\log_b a})$ ;
- 3  $T(n) = \Theta(f(n))$  se  $f(n) = \Omega(n^{\log_b a + \epsilon})$  per  $\epsilon > 0$  e  $af(n/b) \leq cf(n)$  per  $c < 1$  e  $n$  sufficientemente grande.

# Esempio

Applicazione del teorema fondamentale

$$T(n) = \begin{cases} aT(n/b) + f(n) & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases}$$

- 1 Nel caso della ricerca binaria, abbiamo  $T(n) = T(n/2) + O(1)$ . Da cui  $a = 1$ ,  $b = 2$ ,  $f(n) = O(1)$ ; siamo nel secondo caso del teorema, da cui  $T(n) = \Theta(\log n)$ .
- 2 Consideriamo  $T(n) = 9T(n/3) + n$ ; in questo caso  $a = 9$ ,  $b = 3$  e  $f(n) = O(n)$ . Siamo nel primo caso,  $f(n) = O(n^{\log_b a - \epsilon})$  con  $\epsilon = 1$ , da cui  $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$ .

# Analisi di algoritmi ricorsivi

## Numeri di Fibonacci

Ricordiamo la definizione della sequenza di Fibonacci:

$$F_n = \begin{cases} 1 & \text{se } n = 1, 2 \\ F_{n-1} + F_{n-2} & \text{se } n > 2 \end{cases}$$

Consideriamo nuovamente il tempo di esecuzione dell'algoritmo ricorsivo banale per calcolare  $F_n$ , il cui tempo di esecuzione  $T(n)$  soddisfa la relazione di ricorrenza

$$T(n) = \begin{cases} c_1 & \text{se } n = 1, 2 \\ T(n-1) + T(n-2) + c_2 & \text{se } n > 2 \end{cases}$$

Vogliamo produrre un limite inferiore e superiore a  $T(n)$



# Analisi di algoritmi ricorsivi

## Numeri di Fibonacci–limite superiore

**Limite superiore.** Sfruttiamo il fatto che  $T(n)$  è una funzione non decrescente:

$$\begin{aligned}T(n) &= T(n-1) + T(n-2) + c_2 \\ &\leq 2T(n-1) + c_2 \\ &\leq 4T(n-2) + 2c_2 + c_2 \\ &\leq 8T(n-3) + 2^2c_2 + 2c_2 + c_2 \\ &\leq \dots \\ &\leq 2^k T(n-k) + c_2 \sum_{i=0}^{k-1} 2^i \\ &\leq \dots \\ &\leq 2^{n-1} c_3\end{aligned}$$

per una opportuna costante  $c_3$ . Quindi  $T(n) = O(2^n)$ .

# Analisi di algoritmi ricorsivi

Numeri di Fibonacci–limite inferiore

**Limite inferiore.** Sfruttiamo ancora il fatto che  $T(n)$  è una funzione non decrescente:

$$\begin{aligned}T(n) &= T(n-1) + T(n-2) + c_2 \\ &\geq 2T(n-2) + c_2 \\ &\geq 4T(n-4) + 2c_2 + c_2 \\ &\geq 8T(n-6) + 2^2c_2 + 2c_2 + c_2 \\ &\geq \dots \\ &\geq 2^k T(n-2k) + c_2 \sum_{i=0}^{k-1} 2^i \\ &\geq \dots \\ &\geq 2^{\lfloor n/2 \rfloor} c_4\end{aligned}$$

per una opportuna costante  $c_4$ . Quindi  $T(n) = \Omega(2^{\lfloor n/2 \rfloor})$ .

Attenzione  $2^{\lfloor n/2 \rfloor} = O(2^n)$ , ma  $2^{\lfloor n/2 \rfloor} \neq \Theta(2^n)$ . In altre parole, le due funzioni, pur essendo entrambi esponenziali, appartengono a classi di complessità differenti (**Perché?**).

# Costo ammortizzato

L'**analisi ammortizzata** studia il costo **medio** di una sequenza di operazioni.

## Definizione

*Sia  $T(n, k)$  il tempo totale richiesto da un algoritmo, nel caso pessimo, per effettuare  $k$  operazioni su istanze di lunghezza  $n$ . Definiamo il **costo ammortizzato** su una sequenza di  $k$  operazioni come*

$$T_{\alpha}(n) = \frac{T(n, k)}{k}$$

Problema: è data una sequenza di cifre binarie, inizialmente tutte zero. Vogliamo scrivere una funzione che incrementa di uno il valore (decimale) rappresentato dalla sequenza binaria.

```
// v[0] e' il bit piu' significativo  
public static void incrementa( int[] v )  
{  
    for ( int i=v.length-1; i>0; —i ) {  
        v[i] = 1-v[i]; // inverte il bit  
        if ( v[i] == 1 ) {  
            break;  
        }  
    }  
}
```

# Esempio

<b>valore</b>	$v[0]$	$v[1]$	$v[2]$	$v[3]$	$v[4]$	$v[5]$	<b>Costo</b>
0	0	0	0	0	0	0	
1	0	0	0	0	0	<b>1</b>	1
2	0	0	0	0	<b>1</b>	<b>0</b>	2
3	0	0	0	0	1	<b>1</b>	1
4	0	0	0	<b>1</b>	<b>0</b>	<b>0</b>	3
5	0	0	0	1	0	<b>1</b>	1
6	0	0	0	1	<b>1</b>	<b>0</b>	2
7	0	0	0	1	1	<b>1</b>	1
8	0	0	<b>1</b>	<b>0</b>	<b>0</b>	<b>0</b>	4
9	0	0	1	0	0	<b>1</b>	1
10	0	0	1	0	<b>1</b>	<b>0</b>	2

Il costo dell'operazione `inverti` è uguale al numero di bit invertiti.

- Il primo bit ( $v[n-1]$ ) viene invertito ad ogni chiamata;
- Il secondo bit ( $v[n-2]$ ) viene invertito ogni 2 chiamate;
- Il terzo bit ( $v[n-3]$ ) viene invertito ogni 4 chiamate;
- ...
- L' $i$ -esimo bit ( $v[n-i]$ ) viene invertito ogni  $2^{i-1}$  chiamate;

Il tempo totale di  $k$  operazioni è dato da:

$$T(n, k) = k + \lfloor k/2 \rfloor + \lfloor k/4 \rfloor + \dots + 2 + 1 = \sum_{i=0}^{\log_2 k} \lfloor k/2^i \rfloor \leq k \sum_{i=0}^{\infty} 1/2^i = 2k$$

Da cui

$$T_\alpha(n) = \frac{T(n, k)}{k} = O(1)$$