

# Tecniche di analisi degli algoritmi

Damiano Macedonio  
mace@unive.it

Algoritmi e Strutture Dati, A.A. 2014/15

25 ottobre 2014

Original work Copyright ©2009 Moreno Marzolla,  
Università di Bologna

Modifications Copyright ©2014 Damiano Macedonio,  
Università Ca' Foscari di Venezia

This work is licensed under the Creative Commons  
Attribution-NonCommercial-ShareAlike License.

To view a copy of this license, visit

<http://creativecommons.org/licenses/by-nc-sa/3.0/>

or send a letter to Creative Commons, 543 Howard Street, 5th Floor,  
San Francisco, California, 94105, USA.

# Modello di calcolo: Architettura di von Neumann

## Macchina a registri:

- Esiste un dispositivo di input e un dispositivo di output;
- La macchina ha una memoria composta da  $N$  locazioni, con indirizzo da 1 a  $N$ ; ciascuna locazione di memoria può contenere un qualsiasi valore intero o 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;
- Il programma è composto da un insieme **finito** di istruzioni

## Limitazioni del modello:

- Assumiamo che la macchina possa manipolare in tempo costante valori di dimensione **arbitraria**.
- Adottiamo una visione **piatta** della memoria.

# Complessità 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.

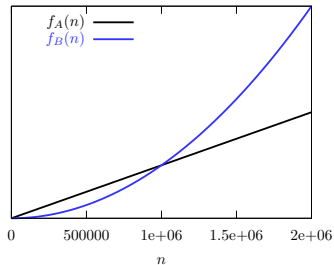
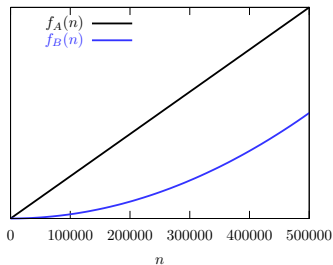
# Complessità computazionale

## Esempio

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

- Sia  $f_A(n) = 10^3 n$  la complessità computazionale di  $A$ ;
- Sia  $f_B(n) = 10^{-3} n^2$  la complessità 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:

$$\forall n \geq n_0 : g(n) \leq c \cdot f(n)$$

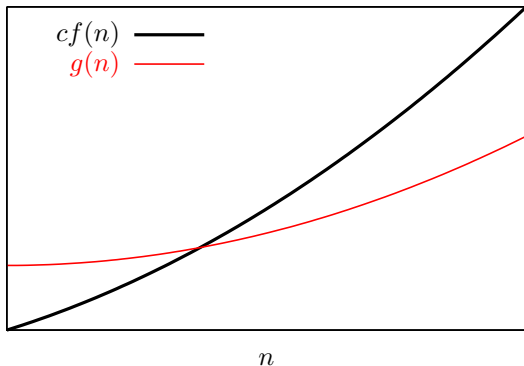
In maniera piú sintetica:

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

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

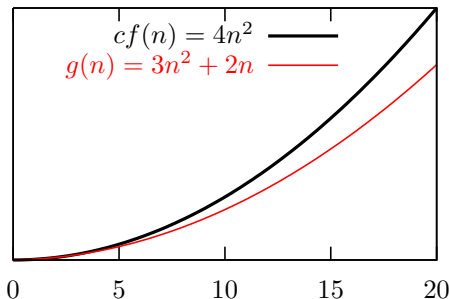
Sia  $g(n) = 3n^2 + 2n$  e  $f(n) = n^2$ . Dimostriamo che  $g(n) = O(f(n))$ .

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:

$$\forall n \geq n_0 : g(n) \geq c \cdot f(n)$$

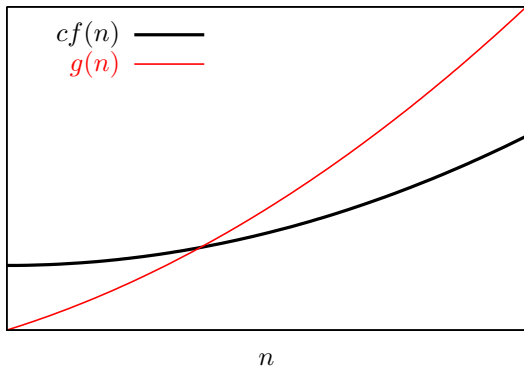
In maniera piú sintetica:

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

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

# Rappresentazione grafica

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



# Esempio

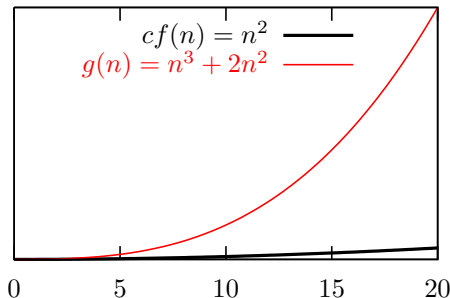
Sia  $g(n) = n^3 + 2n^2$  e  $f(n) = n^2$ , e dimostriamo che  $g(n) = \Omega(f(n))$ .

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

$$n^3 + 2n^2 \geq cn^2 \tag{2}$$

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

se ad esempio scegliamo  $n_0 = 0$  e  $c = 1$ , si ha che la relazione (2) è verificata.



# 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:

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

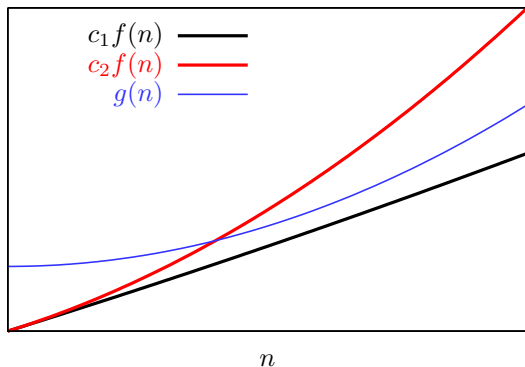
In maniera piú sintetica:

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

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

# Rappresentazione grafica

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



# Alcune proprietà delle notazioni asintotica

## Dualità

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

## 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  $\Theta$ .

# Un metodo di verifica

## Teorema

Siano  $f(n)$  e  $g(n)$  funzioni positive:

- 1 Se  $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = +\infty$  allora  $f(n) = \Omega(g(n))$   
o, equivalentemente,  $g(n) = O(f(n))$ .
- 2 Se  $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = 0$  allora  $f(n) = O(g(n))$   
o, equivalentemente,  $g(n) = \Omega(f(n))$ .
- 3 Se  $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = l \neq 0$  allora  $f(n) = \Theta(g(n))$   
o, equivalentemente,  $g(n) = \Theta(f(n))$ .

# Ordini di grandezza

In ordine di complessità crescente:

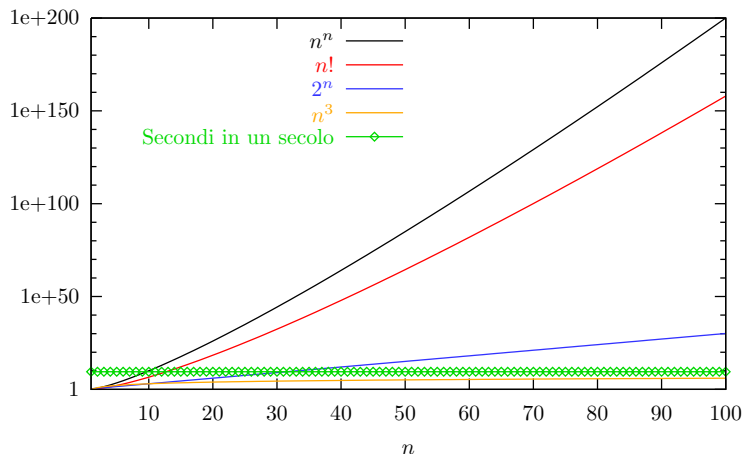
	Ordine	Esempio
$O(1)$	costante	Determinare se un numero è pari
$O(\log n)$	logaritmico	Ricerca di un elemento in un array ordinato
$O(n)$	lineare	Ricerca di un elemento in un array disordinato
$O(n \log n)$	pseudolineare	Ordinamento mediante Merge Sort
$O(n^2)$	quadratico	Ordinamento mediante Bubble Sort
$O(n^3)$	cubico	Prodotto di due matrixi $n \times n$ con definizione
$O(c^n)$	esponenziale, base $c > 1$	
$O(n!)$	fattoriale	Determinante con espansione dei minori
$O(n^n)$	esponenziale, base $n$	

In generale:

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

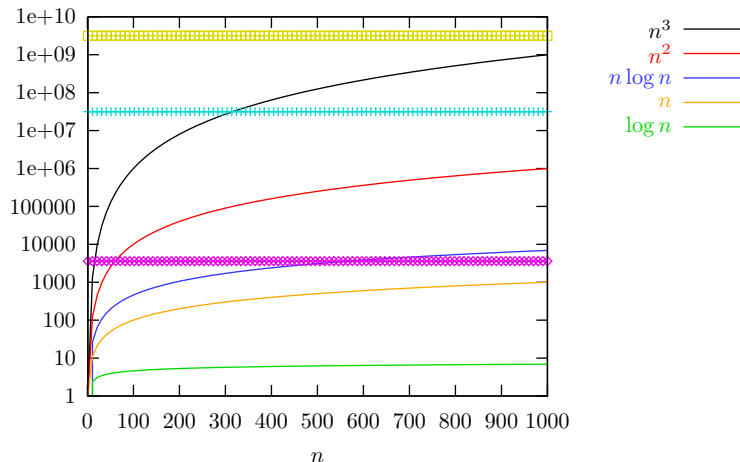


# Confronto grafico tra gli ordini di grandezza



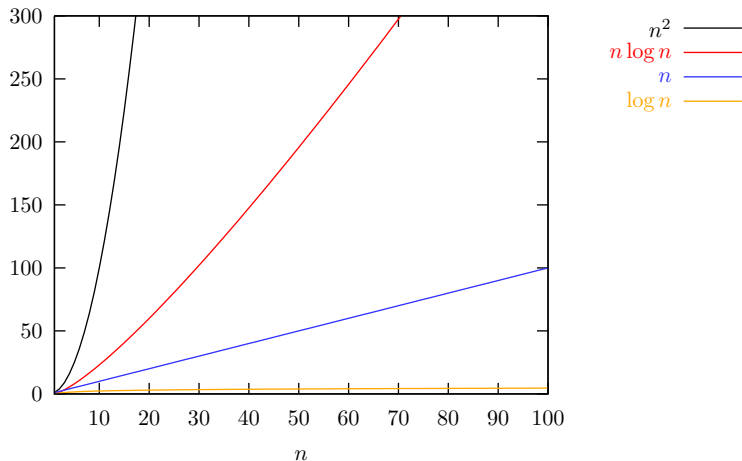
Nota: scala y logaritmica!

# 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



# Domande

- Dimostrare che  $n \log n = O(n^2)$ ;
- Dove collochereste  $O(\sqrt{n})$  nella tabella degli ordini di grandezza? Perché?
- Dimostrare che, per ogni  $\alpha > 0$ ,  $\log n = O(n^\alpha)$ .  
Suggerimento: che cosa potete dire di  $\lim_{n \rightarrow +\infty} \frac{\log n}{n^\alpha}$ ?
- Dimostrare che, per ogni  $a, b > 1$ , vale  $\log_a n = \Theta(\log_b n)$   
Suggerimento: ricordate la regola del cambio di base,  
 $\log_a n = \log_a b \cdot \log_b n$

# Vero o falso?

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

Sia  $F(n) = 6n^2$ . Dobbiamo dimostrare se

$$\exists c > 0, n_0 \geq 0 : \forall n \geq n_0 \quad f(n) \geq cn^3$$

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?

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

Sia  $f(n) = 10n^3 + 2n^2 + 7$ . Dobbiamo dimostrare se

$$\exists c > 0, n_0 \geq 0 : \forall n \geq n_0 \quad f(n) \leq cn^3$$

Possiamo scrivere:

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

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

# 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  $r$  di risorsa **sufficiente** per eseguire  $\mathcal{A}$  su una **qualsiasi istanza** di dimensione  $n$  verifica la relazione  $r(n) = O(f(n))$ .

## Definizione

Un algoritmo  $\mathcal{A}$  ha costo di esecuzione  $\Omega(f(n))$  su istanze di dimensione  $n$  e rispetto ad una certa **risorsa di calcolo**, se la **massima** quantità  $r$  di risorsa **necessaria** per eseguire  $\mathcal{A}$  su istanze di dimensione  $n$  verifica la relazione  $r(n) = \Omega(f(n))$ .

**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  $O(f(n))$ .

## Definizione

Un problema  $\mathcal{P}$  ha **complessità  $\Omega(f(n))$**  rispetto ad una data risorsa di calcolo se **ogni** algoritmo che risolve  $\mathcal{P}$  ha costo di esecuzione  $\Omega(f(n))$  rispetto a quella risorsa.

## Definizione

Dato un problema  $\mathcal{P}$  con **complessità  $\Omega(f(n))$**  rispetto ad una data risorsa di calcolo, un algoritmo che risolve  $\mathcal{P}$  è **ottimo** se ha **costo di esecuzione  $O(f(n))$**  rispetto a quella risorsa.



# 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  $I \in \mathcal{I}_n$ .

- La complessità nel **caso pessimo** (*worst case*) è definita come

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

- La complessità nel **caso ottimo** (*best case*) è definita come

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

- La complessità 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.

# Qualche formula utile...

## Definizione

$\lfloor x \rfloor$  = parte intera di  $x$ , arrotondata per **difetto**

$\lceil x \rceil$  = parte intera di  $x$ , arrotondata per **eccesso**

## Proprietà

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} \quad \text{serie aritmetica}$$

$$\sum_{i=0}^n q^i = \frac{q^{n+1} - 1}{q - 1} \quad \text{serie geometrica di ragione } q \neq 1$$

$$\sum_{i=0}^n i \cdot 2^i = (n-1)2^{n+1} + 2 \quad \text{serie aritmetica-geometrica}$$

# Analisi di algoritmi non ricorsivi

Dato l'array  $A$ , ricerca il valore minimo contenuto in  $A[i], A[i + 1] \dots A[j]$

algoritmo `minimo` (array  $A$ , indici  $i, j$ )  $\rightarrow$  elemento

---

```
// Precondizione:  $i \leq j$  (insieme non vuoto!)
min=i; // Posizione dell'elemento minimo
for k=i+1 to j do
    if ( A[k]<A[min] )
        then min = k
return A[min]
```

---

## Analisi

- Il corpo del ciclo viene eseguito  $j - i$  volte, qualunque sia l'istanza;
- Ogni iterazione ha costo  $O(1)$  (vengono eseguite solo istruzioni elementari).
- Il costo di esecuzione della funzione `minimo` rispetto al tempo è quindi  $\Theta(j - i)$ .
- Sia  $n$  la lunghezza del vettore  $A$ .  
La chiamata `minimo(A, 1, n)` ha tempo di esecuzione  $\Theta(n)$ .

# Osservazione

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

```
if (F_test)
  then
    F_true
  else
    F_false
```

Supponendo:

- $F\_test = O(f(n))$
- $F\_true = O(g(n))$
- $F\_false = O(h(n))$

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

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

# Esempio

Un algoritmo iterativo di ordinamento

algoritmo selectionSort (array  $A$ )

---

```
for i=1 to A.length
  m = minimo( A, i, A.length )
  tmp = A[ i ];
  A[ i ] = A[m];
  A[m] = tmp;
```

---

- Sia  $A$  array di dimensione  $n$ .
- La chiamata `minimo(A, i, A.length)` individua l'elemento minimo tra  $A[i], A[i + 1], \dots, A[n]$  in tempo  $\Theta(n - i)$ .
- L'operazione di scambio ha costo  $O(1)$  in tempo di esecuzione;
- Il corpo del ciclo **for** viene eseguito  $n - 1$  volte.

Il tempo di esecuzione dell'intera procedura `selectionSort` è:

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

che è  $\Theta(n^2)$ .

# Ricerca sequenziale

algoritmo `linearSearch` (*elemento a*, *array A*) → *indice*

---

```
// Trova l'indice della prima occorrenza del valore a nel vettore A.  
// Ritorna 0 se il valore a non e' presente.  
for i=1 to A.length do  
  if ( A[i]==a )  
    then  
      return i  
    else  
      i = i+1  
return 0
```

---

- 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). In tal caso si itera su tutti gli elementi. Quindi  $T_{\text{worst}}(n) = O(n)$
- ... 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 l'elemento cercato sia sempre presente.

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  $i$ .

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

Quindi possiamo concludere che:

$$T_{\text{avg}}(n) = \sum_{i=1}^n P_i \cdot T(i) = \sum_{i=1}^n \frac{1}{n} \cdot i = \frac{1}{n} \sum_{i=1}^n i = \frac{1}{n} \frac{n(n+1)}{2} = O(n)$$

# Analisi di algoritmi ricorsivi

Ricerca di un elemento in un array ordinato

algoritmo `binarySearch` (*elemento a*, *array A*, *indici i, j*) → *indice*

---

```
// Precondizione: A ordinato in senso crescente.
// Trova un'occorrenza di a in A[i],A[i+1],... A[j].
// Ritorna 0 se il valore a non e' presente
if ( i>j )
  then
    return 0
  else
    m = (i+j)/2
    if ( A[m] == a )
      then
        return m
      else
        if ( A[m] > a )
          then
            return binarySearch( a, A, i, m-1 )
          else
            return binarySearch( a, A, m+1, j )
```

---

Per cercare l'elemento *a* in un array *A* di *n* elementi  
invochiamo `binarySearch(a, A, 1, n)`



# Analisi dell'algoritmo di ricerca binaria

Sia  $T(n)$  il tempo di esecuzione della funzione `binarySearch` su una porzione di  $n = j - i + 1$  elementi del vettore  $A$ .

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 (o viene trovato su una porzione di array di dimensione 1). 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:

$$\begin{aligned} T(n) &= T(n/2) + c_2 &= T(n/2) + c_2 \\ &= (T(n/4) + c_2) + c_2 &= T(n/4) + 2c_2 \\ &= ((T(n/8) + c_2) + c_2) + c_2 &= T(n/8) + 3c_2 \\ &\vdots & \\ &= \dots &= T(n/2^i) + i \cdot c_2 \end{aligned}$$

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:  $T(1) = 1 \leq cn$  se  $c \geq 1/n$ . Basta scegliere ad esempio  $c \geq 2$  e questa relazione è verificata per qualsiasi  $n$ ;
- Induzione:

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + n \\ &\leq c\lfloor n/2 \rfloor + n \quad (\text{ipotesi induttiva}) \\ &\leq cn/2 + n = (c/2 + 1)n = f(c)n \end{aligned}$$

La dimostrazione del passo induttivo funziona quando  $f(c) \leq c$ , ossia  $c \geq 2$ .

# Analisi di algoritmi ricorsivi

## Numeri di Fibonacci

Ricordiamo la definizione della sequenza di Fibonacci:

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

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

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

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 ( $T(n-2) \leq T(n-1)$ ):

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + c_2 \\ &\leq 2T(n-1) + c_2 \\ &\leq 2(2T(n-2) + c_2) + c_2 && \text{maggiore } T(n-1) \\ &\leq 2(2(2T(n-3) + c_2) + c_2) + c_2 && \text{maggiore } T(n-2) \\ &\leq 2(2(2(2T(n-4) + c_2) + c_2) + c_2) + c_2 && \text{maggiore } T(n-3) \\ &\vdots \\ &\leq 2^k T(n-k) + \sum_{i=0}^{k-1} 2^i \cdot c_2 \\ &\vdots \\ &\leq 2^n c_1 + \sum_{i=0}^{n-1} 2^i \cdot c_2 = 2^n c_1 + c_2 \frac{2^n - 1}{2 - 1} \end{aligned}$$

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 ( $T(n-1) \geq T(n-2)$ ):

$$\begin{aligned}T(n) &= T(n-1) + T(n-2) + c_2 \\ &\geq 2T(n-2) + c_2 \\ &\geq 2(2T(n-4) + c_2) + c_2 && \text{minoro } T(n-2) \\ &\geq 2(2(2T(n-6) + c_2) + c_2) + c_2 && \text{minoro } T(n-4) \\ &\vdots \\ &\geq 2^k T(n-2k) + \sum_{i=0}^{k-1} 2^i \cdot c_2 \\ &\vdots \\ &\geq 2^{\lfloor n/2 \rfloor} c_1 + \sum_{i=0}^{\lfloor n/2 \rfloor - 1} 2^i \cdot c_2 = 2^{\lfloor n/2 \rfloor} c_1 + c_2 \frac{2^{\lfloor n/2 \rfloor} - 1}{2 - 1}\end{aligned}$$

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?**).

Perché se fosse  $2^{\lfloor n/2 \rfloor} = \Theta(2^n)$  allora dovremmo avere anche  $2^{\lfloor n/2 \rfloor} = \Omega(2^n)$ . Quindi dovrebbero esistere  $c$  ed  $n_0$  tali che  $2^{\lfloor n/2 \rfloor} \geq c 2^n$  per ogni  $n \geq n_0$ . Questo implica  $2^{n/2} \geq c 2^n$  per ogni  $n \geq n_0$ , ovvero

$$\frac{1}{2^{n/2}} \geq c \text{ per ogni } n \geq n_0$$

E questo è impossibile in quanto  $\lim_{n \rightarrow +\infty} \frac{1}{2^{n/2}} = 0$  e quindi è impossibile trovare  $c$  ed  $n_0$  che soddisfino la disuguaglianza.