DIPARTIMENTO DI MATEMATICA E INFORMATICA

UNIVERSITÀ degli STUDI di CATANIA

Complessità

Alessandro Ortis
Image Processing Lab - iplab.dmi.unict.it

alessandro.ortis@unict.it
www.dmi.unict.it/ortis/

IMAGE PROCESSING LABORATORY

Analisi di Complessità

- **Costo di un algoritmo** è in funzione di n (dimensione dei dati in input):
  - **tempo** = numero di **operazioni RAM** eseguite
  - **spazio** = numero di **celle di memoria** occupate (escluse quelle per contenere l'input)

```
if (guardia) {blocco 1} else {blocco 2}
```

```
costo(guardia)+max{costo(blocco 1),costo(blocco 2)}
```

```
for (i=0; i<m; i++) {corpo}
```

$$\sum_{i=0}^{m-1} t_i$$

t_i = costo di `corpo` all'iterazione i

```
while (guardia) {corpo}
do {corpo} while (guardia)
```

$\sum_{i=0}^{m}(t'_i + t_i)$

t'_i = costo di `guardia` all'iterazione i

t_i = costo di `corpo` all'iterazione i

- Il costo di una funzione è dato dal costo del suo corpo (più il passaggio dei parametri)
  - Per le funzioni ricorsive le cose sono più complicate

- Il costo di una sequenza di istruzioni è la somma dei costi delle istruzioni nella sequenza

Caso pessimo e caso medio

Complessità o costo computazionale f(n) in tempo e in spazio di un problema Π:

- **caso pessimo o peggiore** = costo **max** tra **tutte** le istanze di Π aventi dimensioni dei dati pari a n
- **caso medio** = costo **mediato** tra tutte le **istanze** di Π aventi dimensioni pari a n

Complessità temporale

Per un algoritmo, è determinata contando il numero di operazioni aritmetiche e logiche, accesso ai file, letture e scritture in memoria, etc.

I° ipotesi semplificativa:

- Tempo impiegato proporzionale al numero di operazioni eseguite (ciascuna a costo unitario)
- Non ci si riferisce a una specifica macchina.

Complessità temporale

Il tempo impiegato per risolvere un problema dipende sia dall'algoritmo utilizzato sia dalla "dimensione" dei dati a cui si applica l'algoritmo.

Individuare con esattezza una funzione che esprime il tempo impiegato da un algoritmo in funzione della dimensione dell'input *n* è spesso molto difficile.

II° ipotesi semplificativa.

- È sufficiente stabilire il comportamento asintotico della funzione quando le dimensioni dell'ingresso tendono ad infinito (comportamento asintotico dell'algoritmo).
- Si usa a questo scopo la notazione asintotica

Notazione Asintotica

Mediante l'analisi asintotica valutiamo le performance di un algoritmo in termini di dimensioni di input.

Non calcoliamo esplicitamente il tempo (o lo spazio) impiegato da un algoritmo, ma piuttosto come questi parametri crescono al variare della dimensione dell'input.

In altre parole, la complessità dell'algoritmo viene espressa in funzione della dimensione delle strutture dati su cui opera.

Notazione Asintotica ''O-grande''

Denotiamo con O(g(n)) l'insieme delle funzioni

$O(g(n)) = \{ f(n) :$ *esistono delle costanti positive* c, n_0 *t.c.*

$0 \leq f(n) \leq cg(n)$ *per ogni* $n > n_0 \}$

Notazione Asintotica ''O-grande''

Denotiamo con O(g(n)) l'insieme delle funzioni

$$O(g(n)) = \{ f(n) : \text{esistono delle costanti positive } c, n_0 \text{ t.c. } 0 \le f(n) \le cg(n) \text{ per ogni } n > n_0 \}$$

Poiché O(g(n)) è un insieme potremmo scrivere

$$f(n) \in O(g(n))$$

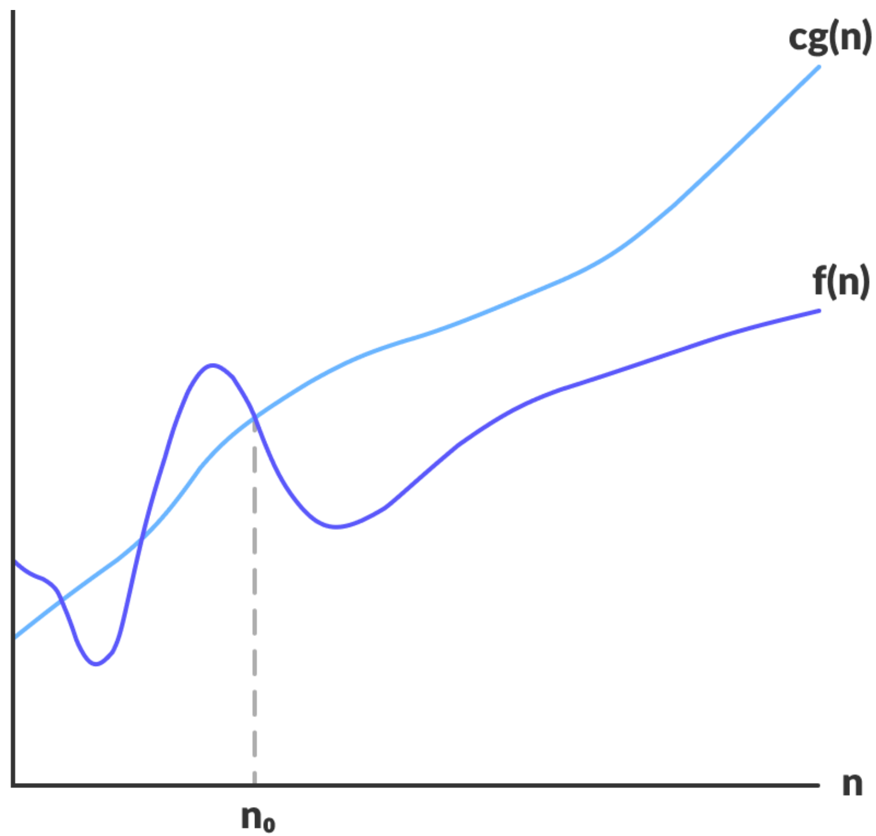
ma di solito scriveremo

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

Notazione Asintotica ‘’O-grande’’

$f(n) = O(g(n))$ sse esistono $c, n_0 > 0$:

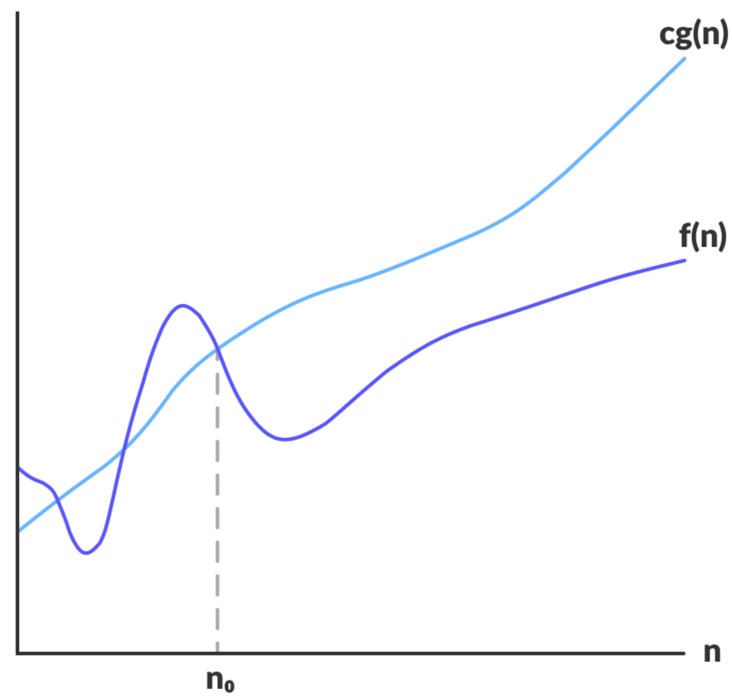
$f(n) \leq c\, g(n)$ per ogni $n > n_0$



Notazione Asintotica ‘’O-grande’’

Quindi, per n sufficientemente grande il tasso di crescita di f(n) è al più proporzionale a g(n).

La notazione O-grande definisce un limite superiore per f(n).



Notazione Asintotica ’’O-grande’’

Attraverso la notazione O( ), gli algoritmi vengono divisi in classi di equivalenza, ponendo nella medesima classe tutti quelli la cui complessità asintotica è dello stesso ordine di grandezza.

Si hanno così algoritmi (funzioni) di complessità asintotica di ordine:

- Costante 1, ...
- Sotto-lineare log n, n^c con $c<1$
- Lineare n
- Polinomiale n*log n, n^2 , n^3 , ... n^c con $c>1$
- Esponenziale c^n, ...n^n, …

Funzione costante

$f(n)=c$ c costante

Funzione logaritmica

$f(n)=\log_b n$ $(b>1)$

1. $x=\log_b (n) \Leftrightarrow b^x=n$
2. $\log_b(1)=0$

Funzione lineare

$f(n)=cn$ c costante (c non nulla)

Funzione nlog n

$f(n)=n \log n$

Funzione esponenziale

$f(n)=b^n$

Regole

- $(b^a)^c=b^{ac}$
- $b^a\ b^c=b^{a+c}$
- $b^a/b^c=b^{a-c}$

Funzione polinomiale

$f(n)=a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d$

il grado è il valore della potenza più grande con a_d diverso da 0

Funzione quadratica

$f(n)=c\ n^2$

```
for (i=0;i<n;i++)
  for(j=0;j<n;j++)
    do something
```

Funzione cubica

$f(n)=c\ n^3$

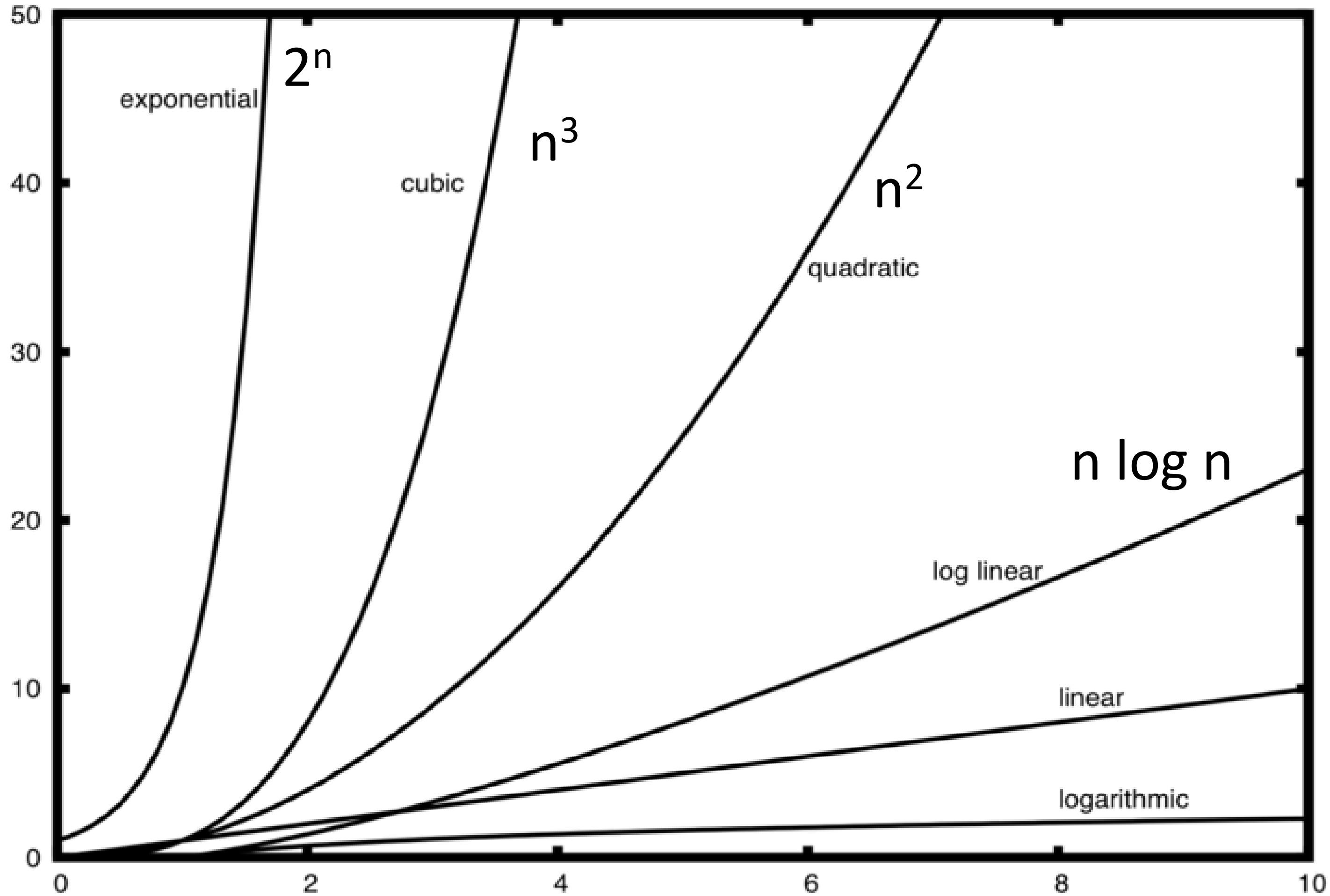
```
for (i=0;i<n;i++)
  for(j=0;j<n;j++)
    for(k=0;k<n;k++)
        do something
```

Notazione Asintotica

Qui di seguito mostriamo alcuni ordini di grandezza tipici, elencati in maniera crescente.

f(n)	
1	
log n	
$\sqrt{n}$	
n	
n log n	
n^c	c>1
c^n	c>1

Notazione Asintotica



Notazione Asintotica ‘’O-grande’’

NB: la notazione O-grande rappresenta ***una*** delimitazione asintotica superiore alla complessità dell'algoritmo, e non ***la*** delimitazione asintotica superiore.

Notazione Asintotica ''O-grande''

NB: la notazione O-grande rappresenta ***una*** delimitazione asintotica superiore alla complessità dell'algoritmo, e non ***la*** delimitazione asintotica superiore.

Infatti, se $T(n) = O(n^4)$, è anche vero che:

$T(n) = O(n^7)$ $\quad$ $T(n) = O(n^4 \log n)$ ecc.

Notazione Asintotica ‘’O-grande’’

NB: la notazione O-grande rappresenta ***una*** delimitazione asintotica superiore alla complessità dell'algoritmo, e non ***la*** delimitazione asintotica superiore.

Infatti, se $T(n) = O(n^4)$, è anche vero che:

$T(n) = O(n^7)$ $T(n) = O(n^4 \log n)$ ecc.

Se per una funzione $T(n)$ sono note più delimitazioni asintotiche superiori, allora è da preferire quella più piccola.

Esempi

$8n \; - 2 = O(n)$ ponendo n_0 =1 e c =6 (o superiore)

Esempi

$8n - 2 = O(n)$ ponendo n_0 =1 e c =6 (o superiore)

$5n^4 + 3n^3 + 2n^2 + 4n + 7 = O(n^4)$ ponendo

c= 5 + 3 + 2 + 4 + 7 = 21

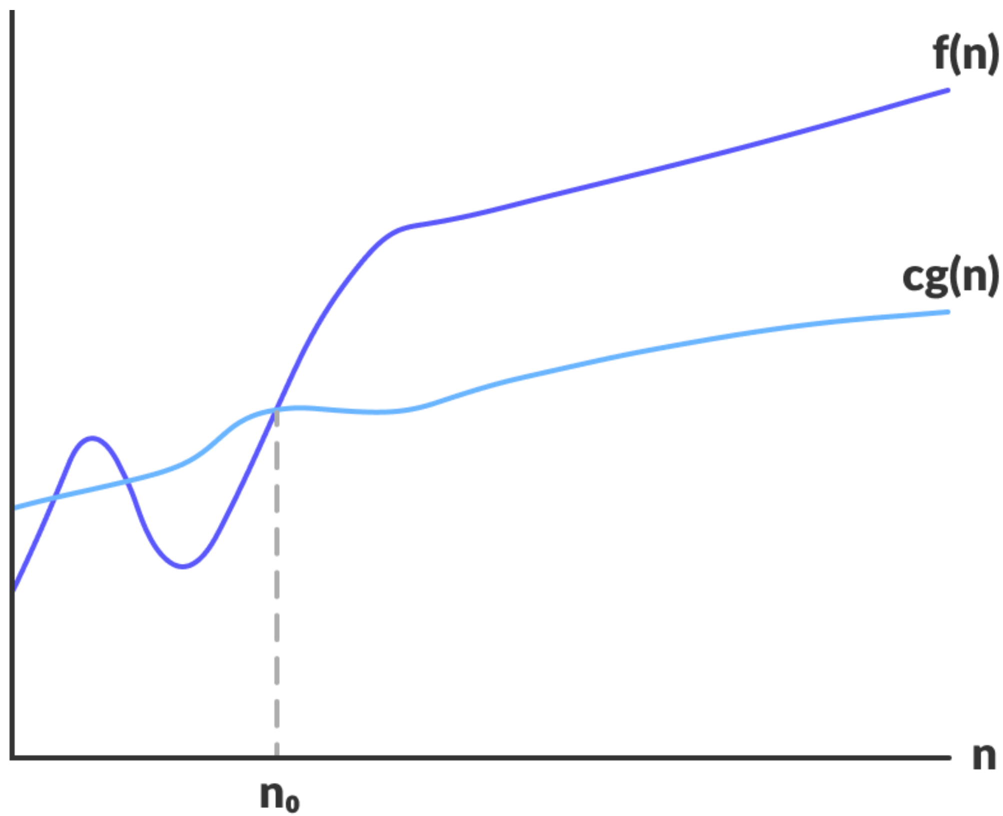
n_0 = 1

$$5n^4 + 3n^3 + 2n^2 + 4n + 7 \leq 21n^4 \ \ \forall n \geq 1$$

Notazione Asintotica "Ω-grande"

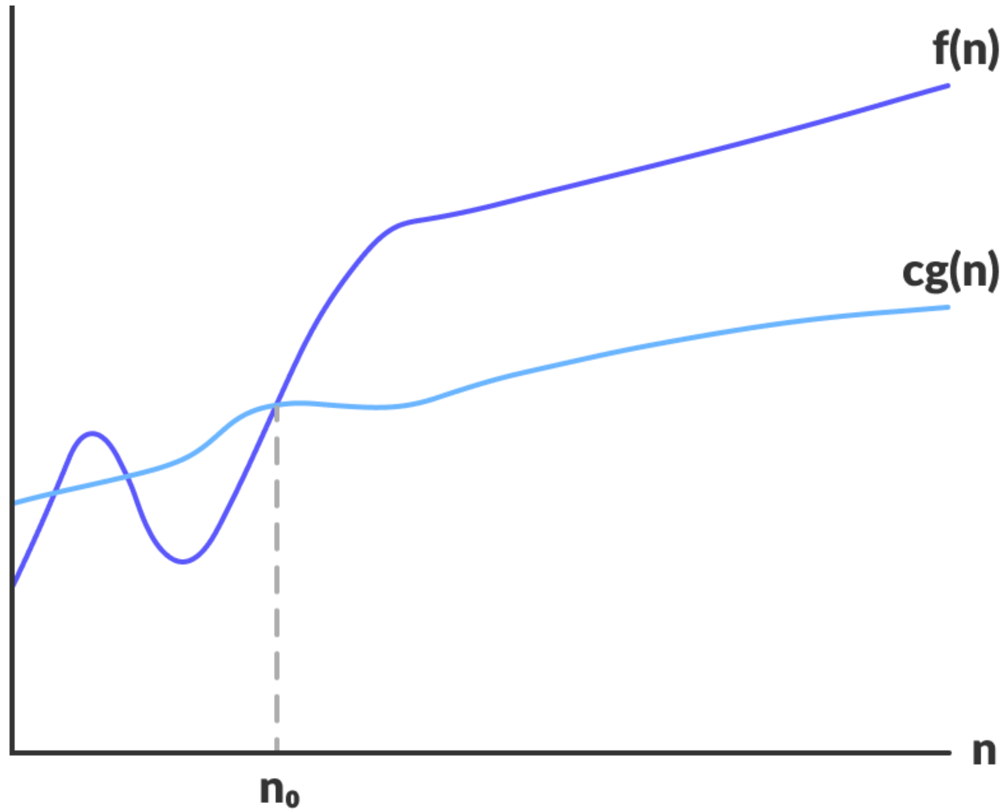
f(n) = Ω(g(n)) sse esistono $c, n_0 > 0$:

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



Notazione Asintotica ''Ω-grande''

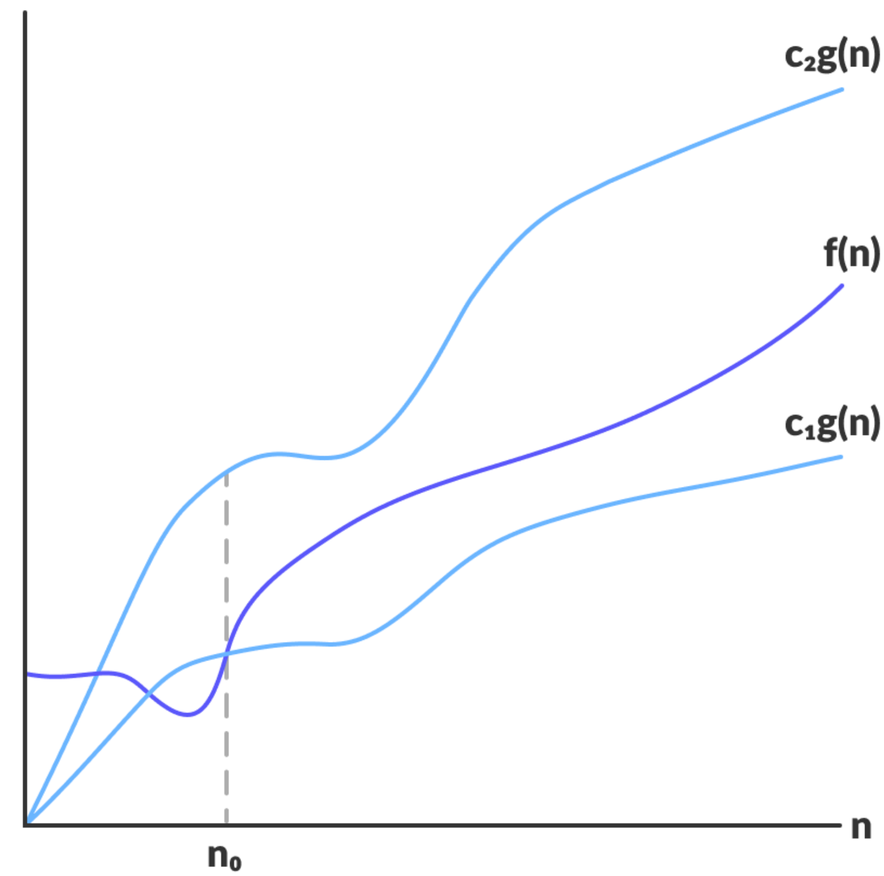
La notazione Ω-grande definisce un limite inferiore per f(n). Per tutti i valori di n maggiori di n_0, il valore di f(n) coincide o sta sopra c g(n).



Notazione Asintotica " Θ-grande"

$f(n) = \Theta(g(n))$ sse esistono delle costanti positive c_1, c_2 ed n_0 t.c. $0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ per ogni $n > n_0$. In altre parole:

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



Notazione Asintotica

$f(n) = O(g(n))$ sse esistono $c, n_0 > 0$:

$f(n) \leq c\, g(n)$ per ogni $n > n_0$

$f(n) = \Omega(g(n))$ sse esistono $c, n_0 > 0$:

$f(n) \geq c\, g(n)$ per ogni $n > n_0$

$f(n) = \Theta(g(n))$ sse **$f(n) = O(g(n))$** e **$g(n) = O(f(n))$**

Notazione Asintotica

1. Se $f(n) = O(g(n))$ allora $a*f(n) = O(g(n))$ $\forall$ costante a

2. Se $f(n) = O(g(n))$ e $g(n) = O(h(n))$ allora $f(n) = O(h(n))$

1) e 2) valgono anche per $\Omega()$ e $\Theta()$

Delimitazioni alla complessità di P

Per un dato problema P consideriamo un algoritmo A che lo risolve.

- Se A prende tempo t(n) diremo che $O(t(n))$ è un **limite superiore.**
- Se riusciamo a provare che nessun algoritmo può far meglio di t(n) diremo che $\Omega(t(n))$ è un **limite inferiore**.
- A è ottimo se i due limiti coincidono
  - In tal caso la complessità computazionale *del problema* è $\Theta(t(n))$.

Delimitazioni alla complessità di P

Un algoritmo A che risolve un problema P è ***ottimale*** se:

1. P ha complessità $\Omega(f(n))$
2. A ha complessità $O(f(n))$

Delimitazioni alla complessità di P

Esempio: il problema dell'ordinamento ha complessità $\Omega(n \log n)$. Quindi un algoritmo di ordinamento con complessità $O(n \log n)$ è ottimale.

Analisi di Algoritmi: esempi pratici

O(1) : complessità di una funzione o blocco di istruzioni ciascuna di costo O(1), che non contengono cicli, ricorsione o chiamate ad altre funzioni non costanti.

O(n) : complessità di un ciclo quando le sue variabili (es. contatore) sono incrementate/decrementate di una quantità costante.

O(n^c) : la complessità di cicli annidati è uguale al numero di volte in cui le istruzioni del ciclo interno vengono eseguite.

Analisi di Algoritmi: esempi pratici

```
// c è una costante positiva
for (int i = 0; i <= n; i += c) {
//espressioni con costo O(1)
}
```

O(n)

```
// c è una costante positiva
for(int i = 1; i <=n; i += c) {
    for (int j = 1; j <=n; j += c) {
     //espressioni con costo O(1)
       }
  }
```

O(n^2)

Analisi di Algoritmi: esempi pratici

O(Logn) : complessità di un ciclo quando le sue variabili sono incrementate/decrementate moltiplicandole/dividendole per una costante.

O(LogLogn) : complessità di un ciclo quando le sue variabili sono incrementate/decrementate esponenzialmente.

Analisi di Algoritmi: esempi pratici

```
// c è una costante positiva
for (int i = 1; i <= n; i *= c) {
   //espressioni con costo O(1)
}
```

O(log n)

```
// c è una costante positiva > 1
for(int i = 2; i <=n; i = pow(i,c)) {
   //espressioni con costo O(1)
}
```

O(log log n)

Esempio

Qual è la complessità del seguente algoritmo?

```
void func(int n) {
int count=0;
for(i=n/2; i<=n; i++) {
    for(j=1; j<=n; j=2*j) {
        for(k=1; k<=n; k=k*2){
            count++; }
} } }
```

Esempio

Qual è la complessità del seguente algoritmo?

```
void func(int n) {
int count=0;
for(i=n/2; i<=n; i++) {
    for(j=1; j<=n; j=2*j) {
        for(k=1; k<=n; k=k*2){
            count++; }
} } }
```

O(n)

Esempio

Qual è la complessità del seguente algoritmo?

```
void func(int n) {
int count=0;
for(i=n/2; i<=n; i++) {
    for(j=1; j<=n; j=2*j) {
        for(k=1; k<=n; k=k*2){
            count++; }
} } }
```

O(n)

O(log n)

Esempio

Qual è la complessità del seguente algoritmo?

```
void func(int n) {
int count=0;
for(i=n/2; i<=n; i++) {
    for(j=1; j<=n; j=2*j) {
        for(k=1; k<=n; k=k*2){
            count++; }
} } }
```

O(n)

O(log n)

O(log n)

Esempio

Qual è la complessità del seguente algoritmo?

```
void func(int n) {
int count=0;
for(i=n/2; i<=n; i++) {
    for(j=1; j<=n; j=2*j) {
        for(k=1; k<=n; k=k*2){
            count++; }
} } }
```

O(n)

O(log n)

O(log n)

$\boxed{\mathbf{O(n \log^2 n)}}$

Esempio

Qual è l'ordine di grandezza (notazione O-grande) della funzione G

$$\mathrm{T1} = 5\mathrm{n}^2 + 2\mathrm{n} - 10$$

$$\mathrm{T2} = 5\sqrt{\mathrm{n}} + 22$$

$$\mathrm{G} = \mathrm{T1} + \mathrm{T2}$$

Esempio

Qual è l'ordine di grandezza (notazione O-grande) della funzione G

$$T1 = 5n^2 + 2n - 10$$

$$T2 = 5\sqrt{n} + 22$$

$$G = T1 + T2 \longrightarrow \boxed{O(n^2)}$$

Esempio

Qual'è la relazione tra T1 e T2? (solo n è variabile) $T1 = O(nlog^2n)$
$T2 = O(1.5nlogn)$

A) T1 > T2

B) T1 < T2

C) T1 = T2

Esempio

Qual'è la relazione tra T1 e T2? (solo n è variabile) $T1 = O(nlog^2 n)$
$T2 = O(1.5nlogn)$

A) T1 > T2

B) T1 < T2

C) T1 = T2

Esempio

Ricerca di un elemento:

v[]	**2**	**5**	**3**	**0**	**12**	**45**	**55**	**12**	**4**
	v[0]	v[1]	…						v[n-1]

```
int ricercaLineare (int vettore[], int dim, int chiave)
{
   for (int i = 0; i < dim; i++)
      if (vettore[i] == chiave) return i;
   return -1;
}
```

Caso migliore: 1
Caso peggiore: n
Caso medio: ?

Esempio

Ricerca di un elemento:

v[]	2	5	3	0	12	45	55	12	4
	v[0]	v[1]	…						v[n-1]

```
int ricercaLineare (int vettore[], int dim, int chiave)
{
   for (int i = 0; i < dim; i++)
      if (vettore[i] == chiave) return i;
   return -1;
}
```

Caso migliore: 1
Caso peggiore: n
Caso medio: ?

$$\sum_{(i=1..N)} Prob(el(i))*i$$

Esempio

Ricerca di un elemento:

v[]	**2**	**5**	**3**	**0**	**12**	**45**	**55**	**12**	**4**
	v[0]	v[1]	…						v[n-1]

```
int ricercaLineare (int vettore[], int dim, int chiave)
{
   for (int i = 0; i < dim; i++)
      if (vettore[i] == chiave) return i;
   return -1;
}
```

Caso migliore: 1
Caso peggiore: n
Caso medio: ?

$$\sum_{(i=1..N)} Prob(el(i))*i = \sum_{(i=1..N)} (1/N)*i$$

Esempio

Ricerca di un elemento:

v[]									
	2	**5**	**3**	**0**	**12**	**45**	**55**	**12**	**4**
	v[0]	v[1]	…						v[n-1]

```
int ricercaLineare (int vettore[], int dim, int chiave)
{
   for (int i = 0; i < dim; i++)
      if (vettore[i] == chiave) return i;
   return -1;
}
```

Caso migliore: 1
Caso peggiore: n
Caso medio: (n+1)/2

$$\sum_{(i=1..N)} Prob(el(i))*i = \sum_{(i=1..N)} (1/N)*i$$