

# Tempo di calcolo, complessità asintotica

March 25, 2021

**Obiettivi:** introdurre le nozioni di tempo di calcolo e di confronto asintotico delle funzioni.

**Argomenti:** caso peggiore, caso migliore e caso medio, notazione asintotica e le sue caratteristiche, complessità di un problema.

Ci interessa studiare gli algoritmi per capire quante risorse utilizzano. Risorse considerate: tempo, memoria, hardware (numero di processori).

Si analizza il tempo di esecuzione per

- capire quanto tempo ci vuole per eseguire un algoritmo,
- caratterizzare la dimensione massima dell'ingresso di un'esecuzione ragionevole,
- scegliere fra i vari algoritmi che risolvono lo stesso problema,
- capire dove e come migliorare un certo algoritmo.

**Metodo di analisi:** calcoliamo il numero di operazioni elementari eseguite. Sotto certe condizioni questa misura è proporzionale al tempo di esecuzione (ed è indipendente dal tipo di computer utilizzato e dal linguaggio di programmazione scelta per l'implementazione).

Esempio visto durante lo studio degli algoritmi quadratici di ordinamento: la tabella riporta il numero di operazioni eseguite da INSERTION-SORT e SELECTION-SORT nel caso migliore e peggiore (le costanti variano caso per caso).

	caso migliore	caso peggiore
INSERTION-SORT	$an + b$	$an^2 + bn + c$
SELECTION-SORT	$an^2 + bn + c$	$an^2 + bn + c$

(Con costanti  $a, b, c$  diversi caso per caso.)

Fissata la dimensione del input, il **numero di operazioni dipende** (quasi sempre) **dal input** stesso!

Denotiamo con  $T'(x)$  il numero di operazioni eseguite da un algoritmo con  $x$  in input.

Numero di operazioni nel **caso migliore con un input di dimensione  $n$** :

$$T_{cm}(n) = \min_{\forall x: |x|=n} T'(x)$$

cioè cerchiamo il minimo considerando tutti i possibili input di dimensione  $n$ .

Numero di operazioni nel **caso peggiore con un input di dimensione  $n$** :

$$T_{cp}(n) = \max_{\forall x: |x|=n} T'(x)$$

cioè cerchiamo il massimo considerando tutti i possibili input di dimensione  $n$ .

Numero di operazioni nel **caso medio con un input di dimensione  $n$** :

$$T_{medio}(n) = \sum_{\forall x: |x|=n} p(x)T'(x)$$

dove  $p(x)$  denota la probabilità che l'input è  $x$  e quindi si calcola la media del numero di operazioni considerando tutti i possibili input di dimensione  $n$ .

# 1 Confronto asintotico di funzioni e il suo utilizzo a caratterizzare la complessità di algoritmi

Indichino  $T_1(n)$  e  $T_2(n)$  il numero di operazioni che due algoritmi devono eseguire per risolvere un certo problema nel caso di un input di dimensione  $n$  (caso migliore, caso peggiore o caso medio). **Per confrontare algoritmi, dobbiamo confrontare tra loro funzioni.**

**Esempio 1.** Consideriamo

$$T_1(n) = \frac{n^2}{100}, \quad T_2(n) = 2n + 300$$

Quale algoritmo conviene utilizzare? Dipende da  $n$ :

$$T_1(n) < T_2(n) \text{ se } n < 300 \text{ e } T_1(n) > T_2(n) \text{ se } n > 300.$$

Il primo algoritmo è conveniente in un numero finito di casi ( $n < 300$ ). In un numero infinito di casi conviene applicare il secondo algoritmo ( $n > 300$ ). (Con  $n = 300$  i due algoritmi equivalgono.)

**Esempio 2.** Assumiamo di riuscire di dimezzare il numero di operazioni nel caso del primo algoritmo, cioè

$$T_1(n) = \frac{n^2}{200}, \quad T_2(n) = 2n + 300$$

Quale algoritmo conviene utilizzare? Dipende sempre da  $n$ :

$$T_1(n) < T_2(n) \text{ se } n < 517 \text{ e } T_1(n) > T_2(n) \text{ altrimenti.}$$

Quindi anche se si migliora notevolmente il primo algoritmo, comunque esso è conveniente in un numero finito di casi ( $n < 517$ ). In un numero infinito di casi conviene applicare il secondo algoritmo ( $n \geq 517$ ). (Non esiste un numero intero  $n$  con il quale i due algoritmi equivalgono.)

**Esempio 3.** Sia  $T(n) = 2^n$  per un certo algoritmo e sia  $D$  la dimensione massima che si riesce a trattare con un certo computer dati i vincoli di tempo. Come cambia la dimensione massima trattabile se si usa un computer 4 volte più veloce? Denotiamo con  $D'$  la nuova dimensione massima. Abbiamo

$$\frac{2^{D'}}{4} = 2^D$$

e quindi

$$D' = \log_2 4 + D = 2 + D$$

Quindi anche usando un computer notevolmente più potente, la dimensione massima che si riesce a trattare si sposta di poco. In questo caso, moltiplicare la velocità per un costante conta poco.

Conviene non considerare le costanti perché

- moltiplicando per una costante la dimensione massima di un problema trattabile cambia poco,
- il tipo di crescita di una funzione non dipende dalla scelta della costante,
- la stima esatta delle costanti è molto difficile in pratica.

La **notazione  $O$**  (detto o grande) si utilizza per descrivere la crescita di una funzione **trascurando costanti moltiplicative e un numero finito di valori.**

**Definizione del  $O$**  (limite superiore):

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

cioè  $f(n)$  è  $O$  di  $g(n)$  se e solo se esistono due costanti  $c > 0, n_0$  tali che per ogni  $n > n_0$  abbiamo  $f(n) \leq cg(n)$ .

$f(n) \in O(g(n))$  significa che  $f(n)$  cresce, a parte un fattore costante ( $c$ ) e trascurando un numero finito di valori (quelli  $\leq n_0$ ), al massimo come la funzione  $g(n)$ .

**Esempio 4.** Sia  $T(n) = an + b$ . Dimostriamo che con qualunque  $a > 0$  e  $b$  si ha  $T(n) \in O(n)$ . Applicando direttamente la definizione del  $O$ , dobbiamo trovare due costanti  $c > 0, n_0$  tali che  $\forall n > n_0. an + b \leq cn$ . Possiamo scegliere qualunque  $c > a$  e  $n_0 = \lceil b/(c - a) \rceil$ .

**Esempio 5.** Sia  $T(n) = a_2n^2 + a_1n + a_0$ . Dimostriamo che con qualunque  $a_2 > 0, a_1, a_0$  si ha  $T(n) \in O(n^2)$ . Applicando direttamente la definizione del  $O$ , dobbiamo trovare due costanti  $c > 0, n_0$  tali che  $\forall n > n_0$  abbiamo  $a_2n^2 + a_1n + a_0 \leq cn^2$ . Possiamo scegliere qualunque  $c > a_2$  e  $n_0 = \lceil x \rceil$  dove  $x$  è la soluzione maggiore dell'equazione  $a_2x^2 + a_1x + a_0 = cx^2$  oppure  $n_0 = 1$  se l'equazione  $a_2x^2 + a_1x + a_0 = cx^2$  non ammette nessuna soluzione reale.

Il seguente teorema è utile per dimostrare facilmente se una certa funzione  $f(n)$  sia  $O(g(n))$  oppure no.

**Teorema:** Se il limite

$$a = \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$$

esiste allora

$$0 \leq a < \infty \iff f(n) \in O(g(n))$$

Utilizzando il teorema precedente è facile dimostrare che

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

e

$$a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 \notin O(n^{k-1}) \text{ se } a_k > 0$$

Per trovare la funzione più semplice  $g(n)$  tale che  $f(n)$  è  $O(g(n))$  bisogna trovare il termine dominante di  $f(n)$  e trascurare i termini minori e le costanti moltiplicativi.

Esempi:

$f(n)$	$\in$ o $\notin$	$O(g(n))$
$k$	$\in$	$O(1)$
$85n + 123$	$\in$	$O(n)$
$2n^2 + n + 10$	$\in$	$O(n^2)$
$\sum_{i=0}^k a_i n^i$ con $a_k > 0$	$\in$	$O(n^k)$
$\sum_{i=0}^k a_i n^i$ con $a_k > 0$	$\notin$	$O(n^h)$ se $h < k$
$5n + \log_2 n + 5$	$\in$	$O(n)$
$n + 25 \log_2 n + 5$	$\notin$	$O(\log_2 n)$

**Esempio 6. Base dei logaritmi.** Cambiare la base dei logaritmi corrisponde a moltiplicare per una costante:

$$\log_a n = \frac{1}{\log_b a} \log_b n$$

Di conseguenza si ha

$$\log_a n \in O(\log_b n) \quad \forall a > 1, \forall b > 1$$

Per questo motivo **quando si usa la notazione  $O$  non si indica la base dei logaritmi, si scrive  $O(\log n)$ .**

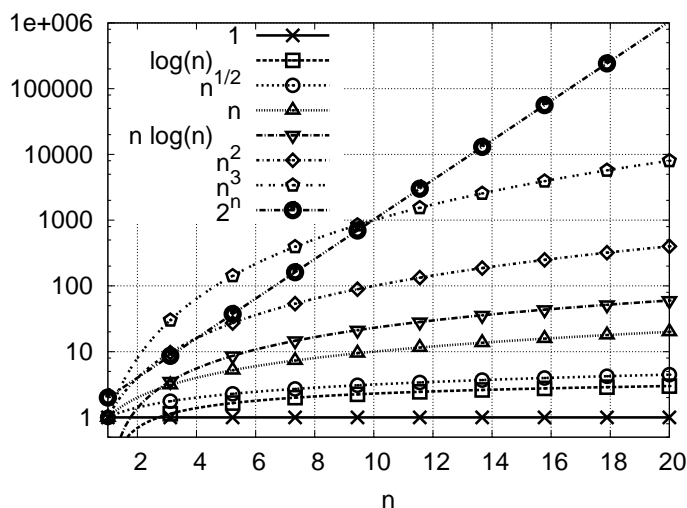
La notazione  $O$  permette di confrontare le funzioni tra loro. La relazione  $\lesssim$  definita come

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

è un **preordine basato sulla notazione  $O$** . Un preordine è una relazione riflessiva, cioè  $f(n) \lesssim f(n)$ , ed è transitiva, cioè  $f(n) \lesssim g(n) \wedge g(n) \lesssim h(n) \implies f(n) \lesssim h(n)$ . Ma non permette di mettere in relazione

qualunque coppia di funzioni. Per esempio, con  $f(n) = \sin(n) + 2, g(n) = \cos(n) + 2$  non si ha ne  $f(n) \lesssim g(n)$  ne  $g(n) \lesssim f(n)$ .

Si possono quindi ordinare le funzioni per velocità di crescita:



La notazione  $O$  è una **notazione insiemistica**. L'insieme  $O(g(n))$  indica l'insieme di funzioni che sono  $O$  di  $g(n)$ . Per esempio, l'insieme  $O(n)$  indica l'insieme di tutte le funzioni che crescono al massimo linearmente. Quindi le funzioni  $k, \log_2 n, \sqrt{n}$  fanno parte dell'insieme  $O(n)$ .

In questa ottica insiemistica, il preordine stabilito dal  $O$  dà luogo ad una **sequenza di inclusioni**. Per esempio,

$$O(1) \subset O(\log n) \subset O(n^{1/2}) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^2 \log n) \subset O(n^3) \subset O(2^n)$$

Spesso la notazione  $O$  viene utilizzata in maniera informale:

$$2n^2 + 6n = O(n^2) \quad 3n^2 = O(n^2)$$

il che, presa alla lettera implicherebbe  $2n^2 + 6n = 3n^2$  che non è vero.  $f(n) = O(g(n))$  si interpreta come un'equazione "a senso unico", con cui rimpiazziamo una funzione con il suo ordine di grandezza.

Per caratterizzare in senso  $O$  la complessità di algoritmi è conveniente utilizzare le seguenti regole:

$$\begin{aligned} f(n) &\in O(f(n)) \\ cf(n) &\in O(f(n)) \quad c \text{ deve essere costante} \\ f(n) + f(n) &\in O(f(n)) \\ f(n) + g(n) &\in O(f(n) + g(n)) \\ f(n) \in O(g(n)) &\implies (f(n) + g(n) \in O(g(n))) \\ f(n)g(n) &\in O(f(n)g(n)) \end{aligned}$$

Con la notazione informale le regole diventano

$$\begin{aligned} f(n) &= O(f(n)) \\ cf(n) &= O(f(n)) \quad c \text{ deve essere costante} \\ f(n) + O(f(n)) &= O(f(n)) \\ f(n) + O(g(n)) &= O(f(n) + g(n)) \\ f(n) = O(g(n)) &\implies (f(n) + O(g(n)) = O(g(n))) \\ f(n)O(g(n)) &= O(f(n)g(n)) \end{aligned}$$

Nota bene  $nf(n) \notin O(f(n))$  perché  $n$  non è costante.

**Esempio 7.** Caratterizziamo in senso  $O$  la complessità del seguente algoritmo.

```

1: ALG( $n$ )
2:  $s \leftarrow 0$ 
3: for  $i \leftarrow 1$  to  $2n$  do
4:    $j \leftarrow 1$ 
5:   while  $j \leq n$  do
6:      $j \leftarrow j + 1$ 
7:      $s \leftarrow s + j$ 
8:    $j \leftarrow 1$ 
9:   while  $j \leq n$  do
10:     $j \leftarrow 2j$ 
11:     $s \leftarrow s + ij$ 
12: return  $s$ 

```

Contiamo il numero di operazioni  $T(n)$  considerando ogni riga come singola operazione.

$$\begin{aligned}
T(n) = & \underbrace{1}_{r.2} + \underbrace{2n+1}_{r.3} + 2n \left( \underbrace{1}_{r.4} + \underbrace{n+1}_{r.5} + n \left( \underbrace{1}_{r.6} + \underbrace{1}_{r.7} \right) + \underbrace{1}_{r.8} + \right. \\
& \left. \underbrace{\lfloor \log_2 n \rfloor + 2}_{r.9} + (\lfloor \log_2 n \rfloor + 1) \left( \underbrace{1}_{r.10} + \underbrace{1}_{r.11} \right) \right) + \underbrace{1}_{r.12}
\end{aligned}$$

Applicando le regole si arriva a stabilire che  $T(n) \in O(n^2)$ .

Con  $O$  si può dare un limite superiore per la velocità della crescita di una funzione. Questo limite non è necessariamente stretto: per esempio,  $\sqrt{n} \in O(2^n)$ . Per poter definire limiti stretti e limiti inferiori si introducono  $\Theta$  (theta grande) e  $\Omega$  (omega grande).

**Definizione del  $\Theta$**  (limiti stretti):

$$f(n) \in \Theta(g(n)) \iff \exists c_1 > 0, \exists c_2 > 0, \exists n_0, \forall n > n_0. c_1 g(n) \leq f(n) \leq c_2 g(n)$$

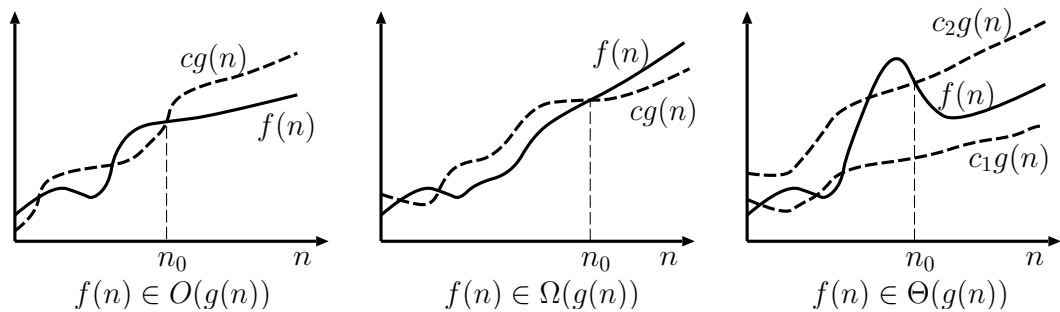
cioè  $f(n)$  è  $\Theta$  di  $g(n)$  se e solo se esistono tre costanti  $c_1 > 0, c_2 > 0, n_0$  tali che per ogni  $n > n_0$  abbiamo  $c_1 g(n) \leq f(n) \leq c_2 g(n)$ .

**Definizione del  $\Omega$**  (limite inferiori):

$$f(n) \in \Omega(g(n)) \iff \exists c > 0, \exists n_0, \forall n > n_0. f(n) \geq cg(n)$$

cioè  $f(n)$  è  $\Omega$  di  $g(n)$  se e solo se esistono due costanti  $c > 0, n_0$  tali che per ogni  $n > n_0$  abbiamo  $f(n) \geq cg(n)$ .

Graficamente  $O$ ,  $\Omega$  e  $\Theta$ :



Segue direttamente dalle definizioni che

$$f(n) \in \Omega(g(n)) \wedge f(n) \in O(g(n)) \iff f(n) \in \Theta(g(n))$$

Come  $O$ , anche  $\Omega$  definisce una relazione che è un preordine:

$$f(n) \succeq g(n) \iff f(n) \in \Omega(g(n))$$

La notazione  $\Theta$  fornisce invece una relazione di equivalenza:

$$f(n) \sim g(n) \iff f(n) \in \Theta(g(n))$$

è riflessiva, transitiva e anche simmetrica ( $f(n) \in \Theta(g(n)) \iff g(n) \in \Theta(f(n))$ ).

Il teorema precedente esteso per  $\Theta$  e  $\Omega$ :

**Teorema:** Se il limite

$$a = \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$$

esiste allora

$$\begin{aligned} 0 \leq a < \infty &\iff f(n) \in O(g(n)) \\ 0 < a < \infty &\iff f(n) \in \Theta(g(n)) \\ 0 < a \leq \infty &\iff f(n) \in \Omega(g(n)) \end{aligned}$$

Riassumiamo qui di seguito alcune proprietà utili degli ordini di grandezza:

1. riflessività:  $f(n) \in O(f(n))$ ; vale anche per  $\Omega$ ,  $\Theta$
2. transitività:  $g(n) \in O(f(n)) \wedge f(n) \in O(h(n)) \implies g(n) \in O(h(n))$ ; vale anche per  $\Omega$ ,  $\Theta$
3. simmetria:  $g(n) \in \Theta(f(n)) \iff f(n) \in \Theta(g(n))$ ; vale solo per  $\Theta$
4. trasposizione:  $g(n) \in O(f(n)) \iff f(n) \in \Omega(g(n))$
5. somma:  $f(n) + g(n) \in O(\max\{f(n), g(n)\})$ ; vale anche per  $\Omega$ ,  $\Theta$
6. prodotto:  $g(n) \in O(f(n)) \wedge h(n) \in O(q(n)) \implies g(n)h(n) \in O(f(n)q(n))$ ; vale anche per  $\Omega$ ,  $\Theta$ .

## 2 Complessità di un problema

Dato un problema si vuole caratterizzare quante operazioni servono per risolverlo.

Qual è un tempo di calcolo **sufficiente** alla risoluzione di un problema?

**Un confine superiore alla complessità di un problema è un confine superiore** per il tempo di calcolo (nel caso peggiore) **di un algoritmo** che risolve il problema.

Siano dati un problema  $P$  e un algoritmo  $A$  che lo risolve. Se il numero di operazioni richieste da  $A$  nel caso peggiore è  $O(g(n))$  allora  $g(n)$  è un confine superiore alla complessità di  $P$ .

Qual è un tempo di calcolo **necessario** alla risoluzione di un problema?

**Un confine inferiore alla complessità di un problema è un confine inferiore** per i tempi di calcolo (nel caso peggiore) **di tutti gli algoritmi** che risolvono il problema.

Sia dato un problema  $P$ . Se il numero di operazioni richieste da qualunque algoritmo  $A$  che risolve  $P$  nel caso peggiore è  $\Omega(g(n))$  allora  $g(n)$  è un confine inferiore alla complessità di  $P$ .

Esistono confini inferiori banali:

- **Dimensione dei dati:** è spesso necessario esaminare tutti i dati in ingresso (oppure generare tutti i dati in uscita). Per esempio, la moltiplicazione di due matrici quadrate di ordine  $n$  richiede l'ispezione di  $2n^2 \in \Omega(n^2)$  numeri.

- **Eventi contabili:** spesso c'è un evento la cui ripetizione un numero di volte è necessaria alla soluzione del problema. Per esempio, la determinazione del massimo tra  $n$  elementi richiede  $n - 1 \in \Omega(n)$  confronti.

Se una funzione  $g(n)$  rappresenta **sia un confine inferiore sia un confine superiore** per un certo problema, allora la **complessità del problema** è  $g(n)$ .

Sia dato un problema e un confine inferiore alla sua complessità  $g(n)$ . Un algoritmo che risolve il problema nel caso peggiore con complessità  $O(g(n))$ , è un **algoritmo ottimo**.

**Esempio 8.** Il problema “somma 17”: dato un vettore di  $n$  interi positivi decidere se ne contiene due la cui somma sia 17.

Si può risolvere il problema con un algoritmo che calcola la somma di tutte le coppie di numeri. Tale algoritmo è quadratico e quindi la complessità del problema è  $O(n^2)$  (un confine superiore).

Qualunque algoritmo che risolve il problema, nel caso peggiore, deve considerare ogni numero del vettore almeno una volta. Quindi la complessità del problema è  $\Omega(n)$  (un confine inferiore).

Un algoritmo migliore:

1. calcoliamo un vettore  $c$  che indica quali numeri fra 0 e 17 (compresi) sono presenti nel vettore:  $\forall i, 0 \leq i \leq 17, c[i] = 1$  se  $i$  è presente nel vettore in input e 0 altrimenti;
2. dato che il vettore in input ha solo interi positivi, la risposta sarà true se e solo se esistono  $i$  e  $j$  tali che  $0 \leq i \leq 17 \wedge 0 \leq j \leq 17 \wedge c[i] = 1 \wedge c[j] = 1 \wedge i + j = 17$ .

Generare  $c$  richiede percorrere il vettore in input una volta e quindi ha complessità  $O(n)$ . Dato  $c$ , il secondo passo può essere risolto con un ciclo for:

```

for  $i \leftarrow 0$  to 8 do
  if  $c[i] = 1 \wedge c[17 - i] = 1$  then
    return true
  end if
end for
return false

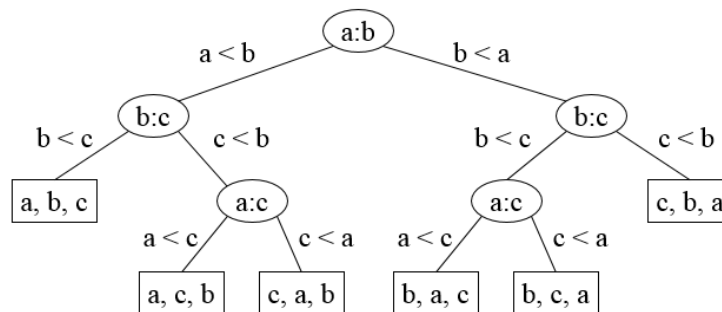
```

Il ciclo richiede un numero costante di operazioni, ha complessità  $O(1)$ . Quindi in totale l'algoritmo ha complessità  $O(n)$ .

Visto che la complessità del problema è  $\Omega(n)$ , l'algoritmo che ha complessità  $O(n)$  è un algoritmo ottimo.

## 2.1 Complessità del problema dell'ordinamento per confronti

I confronti che si devono fare per ordinare un vettore si possono rappresentare con alberi binari di decisione in cui i nodi interni rappresentano i confronti e le foglie i possibili ordini finali. Il seguente albero rappresenta i confronti necessari per ordinare un vettore di tre elementi.



Tale albero, per un vettore di  $n$  elementi, ha  $n!$  foglie (tutte le possibili permutazioni devono esserci).

Con  $n!$  foglie, il ramo più lungo contiene almeno  $\lceil \log_2 n! \rceil$  nodi interni. (Nel caso di  $n = 3$ , abbiamo  $\lceil \log_2 3! \rceil = 3$ .)

Ciò significa che un algoritmo di ordinamento per confronti deve fare nel caso peggiore almeno  $\lceil \log_2 n! \rceil$  confronti.

Secondo la formula di Stirling, per grandi valori di  $n$ , si può approssimare  $n!$  con

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

Dopo qualche algebra segue che, per grandi valori di  $n$ , **il numero di confronti che un algoritmo di ordinamento per confronti deve fare nel caso peggiore è proporzionale a**

$$n \log_2 n$$

Quindi la complessità del problema dell'ordinamento per confronti è almeno  $n \log n$ . Visto che si conoscono algoritmi che risolvono il problema del problema dell'ordinamento per confronti in tempo  $O(n \log n)$ , la complessità di questo problema è  $n \log n$ .