Complessità Computazionale

Introduzione

Un problema di conteggio

- **Input**
  - **Un intero *N* dove *N* ≥ 1.**
- **Output**
  - **Il numero di coppie ordinate (*i* , *j*) tali che *i* e *j* sono interi e 1≤ *i* ≤ *j* ≤ *N*.**

- Esempio: N=4
  - (1,1), (1,2), (1,3), (1,4), (2,2), (2,3),(3,3), (2,4), (3,4), (4,4)
  - Output = 10

Algoritmo 1

```
int Count_1(int N)
    sum = 0
    for i = 1 to N
        for j = i to N
            sum = sum + 1
    return sum
```

`sum = 0`	1
`for i = 1 to N`	$2N$
`for j = i to N`	$2\sum_{i=1}^{N}(N+1-i)$
`sum = sum + 1`	$\sum_{i=1}^{N}(N+1-i)$
`return sum`	1

Il tempo di esecuzione è $\boxed{2+2N+3\sum_{i=1}^{N}(N+1-i)=\frac{3}{2}N^2+\frac{7}{2}N+2}$

Algoritmo 2

```
int Count_2(int N)
    sum = 0
    for i = 1 to N
       sum = sum + (N+1-i)
    return sum
```

`sum = 0`	1
`for i = 1 to N`	$2N$
`sum = sum + (N+1-i)`	$4N$
`return sum`	1

Il tempo di esecuzione è $5N+2$

Ma osserviamo che:

$$\boxed{\sum_{i=1}^{N}(N+1-i)=\sum_{i=1}^{N} i = N(N+1)/2}$$

Algoritmo 3

$$\sum_{i=1}^{N}(N+1-i)=\sum_{i=1}^{N}i=N(N+1)/2$$

```
int Count_3(int N)
1     sum = N(N+1)/2      4
2     return sum          1
```

Il tempo di esecuzione è 5 unità di tempo

Riassunto dei tempi di esecuzione

Algoritmo	**Tempo di Esecuzione**
Algoritmo 2	$\frac{3}{2}N^2 + \frac{7}{2}N + 2$
Algoritmo 3	$\mathbf{6}N\mathbf{+2}$
Algoritmo 4	**5**

Ordine dei tempi di esecuzione

Supponiamo che 1 operazone atomica impieghi 1 μs = 10^{-9} s

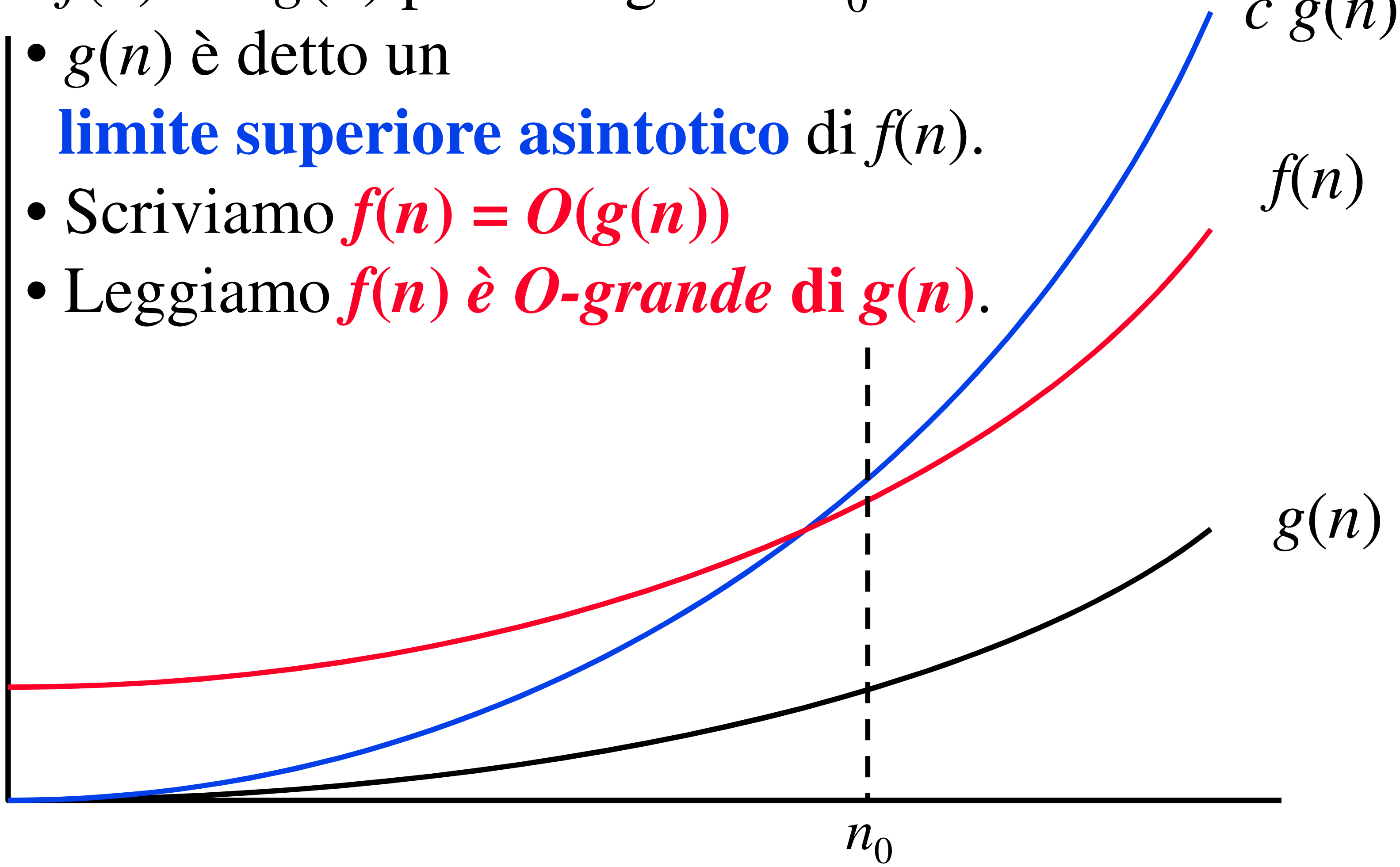
	1000	10000	100000	1000000	10000000
N	1 μs	10 μs	100 μs	1 ms	10 ms
20N	20 μs	200μs	2 ms	20 ms	200 ms
N Log N	9.96 μs	132 μs	1.66 ms	19.9 ms	232 ms
20N Log N	199μs	2.7 ms	33 ms	398 ms	4.6 sec
N^2	1 ms	100 ms	10 sec	17 min	1.2 giorni
$20N^2$	20 ms	2 sec	3.3 min	5.6 ore	23 giorni
N^3	1 sec	17 min	12 gior.	32 anni	32 millenni

Riassunto dei tempi di esecuzione

Algoritmo	Tempo di Esecuzione	Ordine del Tempo di Esecuzione
Algoritmo 1	$5N^2+2N+2$	N^2
Algoritmo 2	$\frac{3}{2}N^2+\frac{7}{2}N+2$	N^2
Algoritmo 3	$6N+2$	N
Algoritmo 4	5	Costante

Limite superiore asintotico

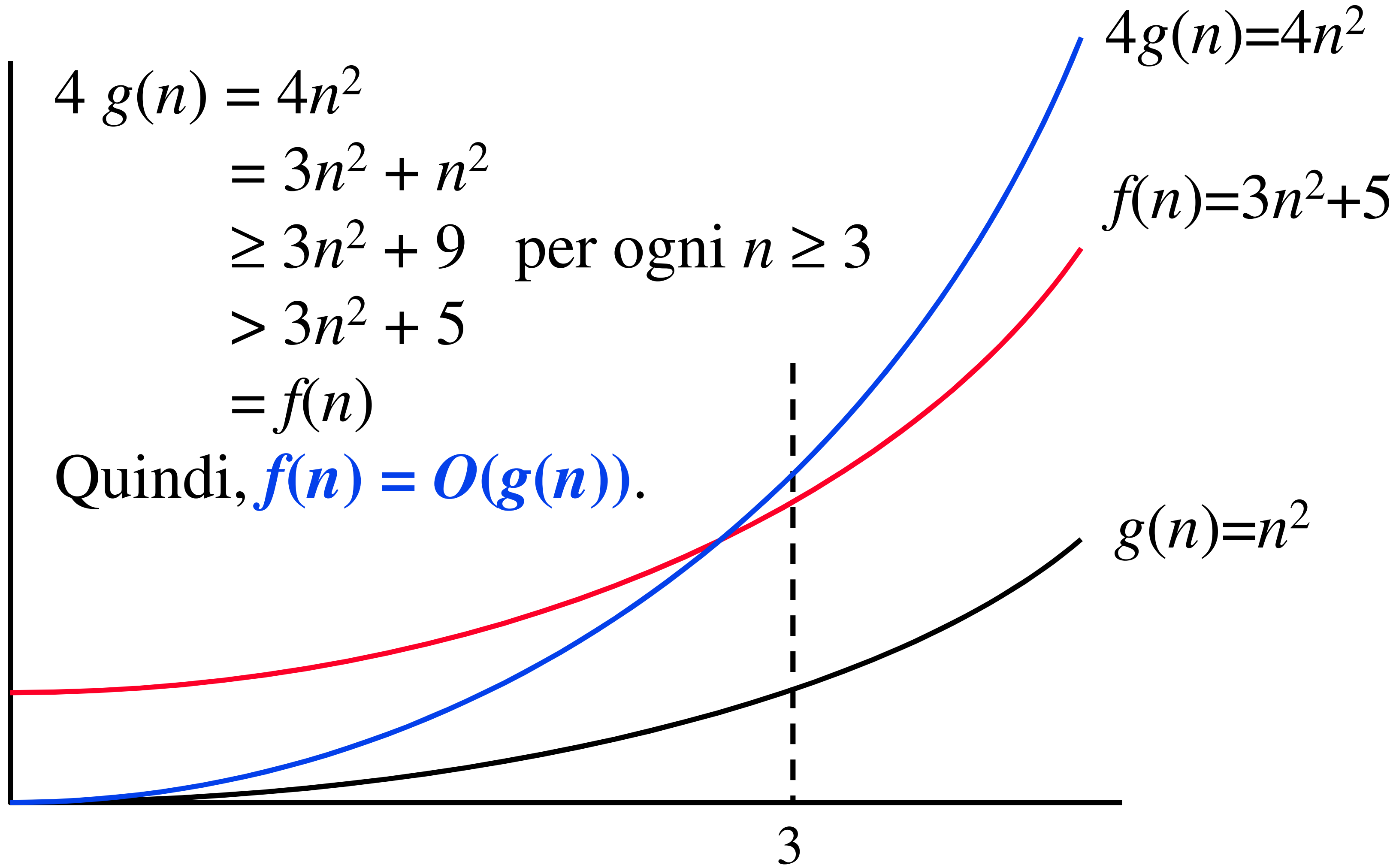
- $f(n) \leq c\ g(n)$ per tutti gli $n \geq n_0$
- $g(n)$ è detto un **limite superiore asintotico** di $f(n)$.
- Scriviamo ***f(n) = O(g(n))***
- Leggiamo ***f(n) è O-grande* di *g(n)***.



Esempio di limite superiore asintotico

$$
\begin{aligned}
4\,g(n) &= 4n^2 \\
&= 3n^2 + n^2 \\
&\geq 3n^2 + 9 \quad \text{per ogni } n \geq 3 \\
&> 3n^2 + 5 \\
&= f(n)
\end{aligned}
$$

Quindi, $\boldsymbol{f(n) = O(g(n))}$.



Esercizio sulla notazione O

- **Mostrare che $3n^2+2n+5 = O(n^2)$**

$$\begin{aligned} 10n^2 &= 3n^2 + 2n^2 + 5n^2 \\ &\geq 3n^2 + 2n + 5 \text{ per } n \geq 1 \end{aligned}$$

$$c = 10,\ n_0 = 1$$

Utilizzo della notazione O

- **In genere quando impieghiamo la notazione *O*, utilizziamo la formula più "*semplice*".**
  - **Scriviamo**
    - $3n^2+2n+5 = O(n^2)$
  - **Le seguenti sono tutte corrette ma in genere non le si usera:**
    - $3n^2+2n+5 = O(3n^2+2n+5)$
    - $3n^2+2n+5 = O(n^2+n)$
    - $3n^2+2n+5 = O(3n^2)$

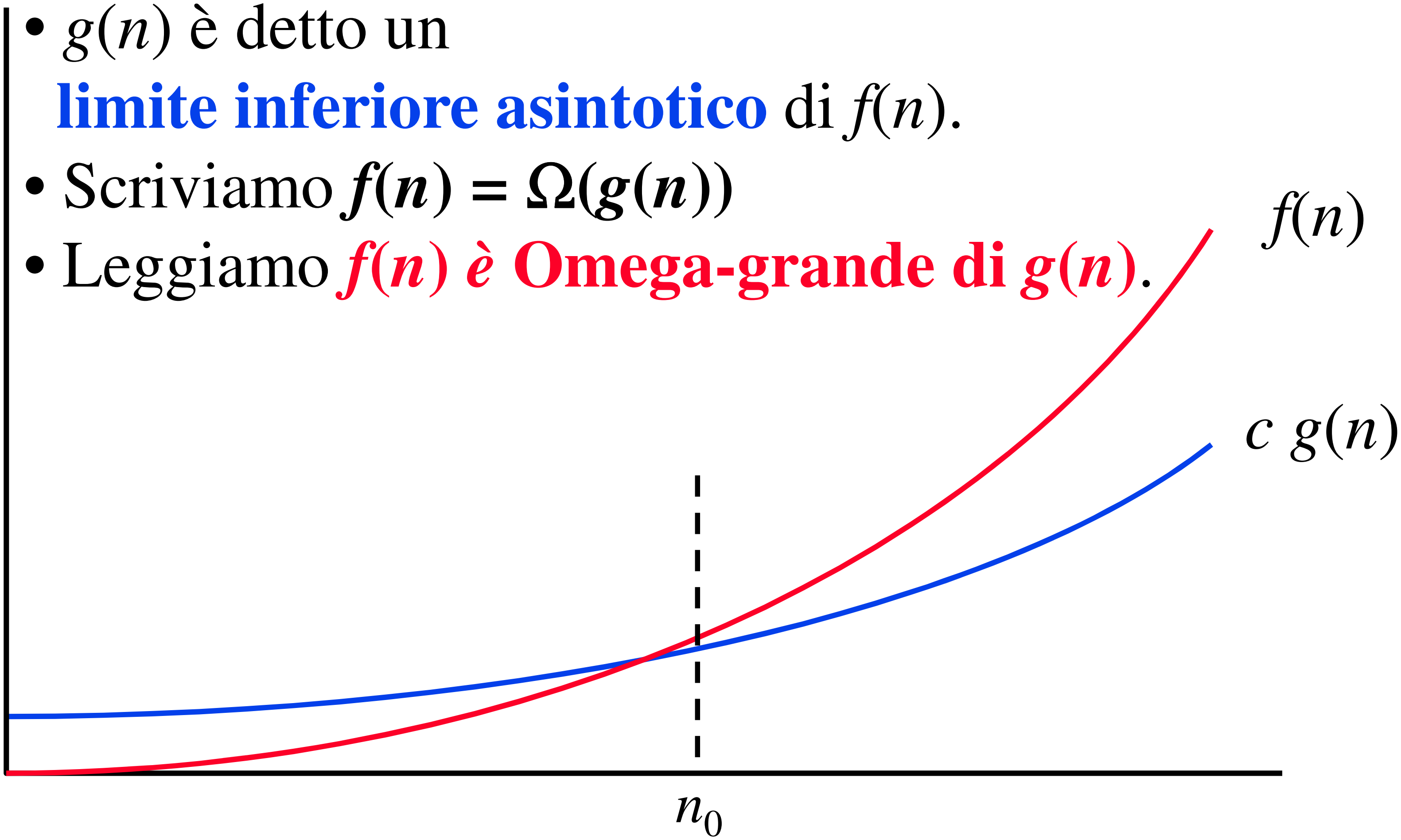
Esercizi sulla notazione O

- $f_1(n) = 10\ n + 25\ n^2$
- $f_2(n) = 20\ n \log n + 5\ n$
- $f_3(n) = 12\ n \log n + 0.05\ n^2$
- $f_4(n) = n^{1/2} + 3\ n \log n$

- $O(n^2)$
- $O(n \log n)$
- $O(n^2)$
- $O(n \log n)$

Limite inferiore asintotico

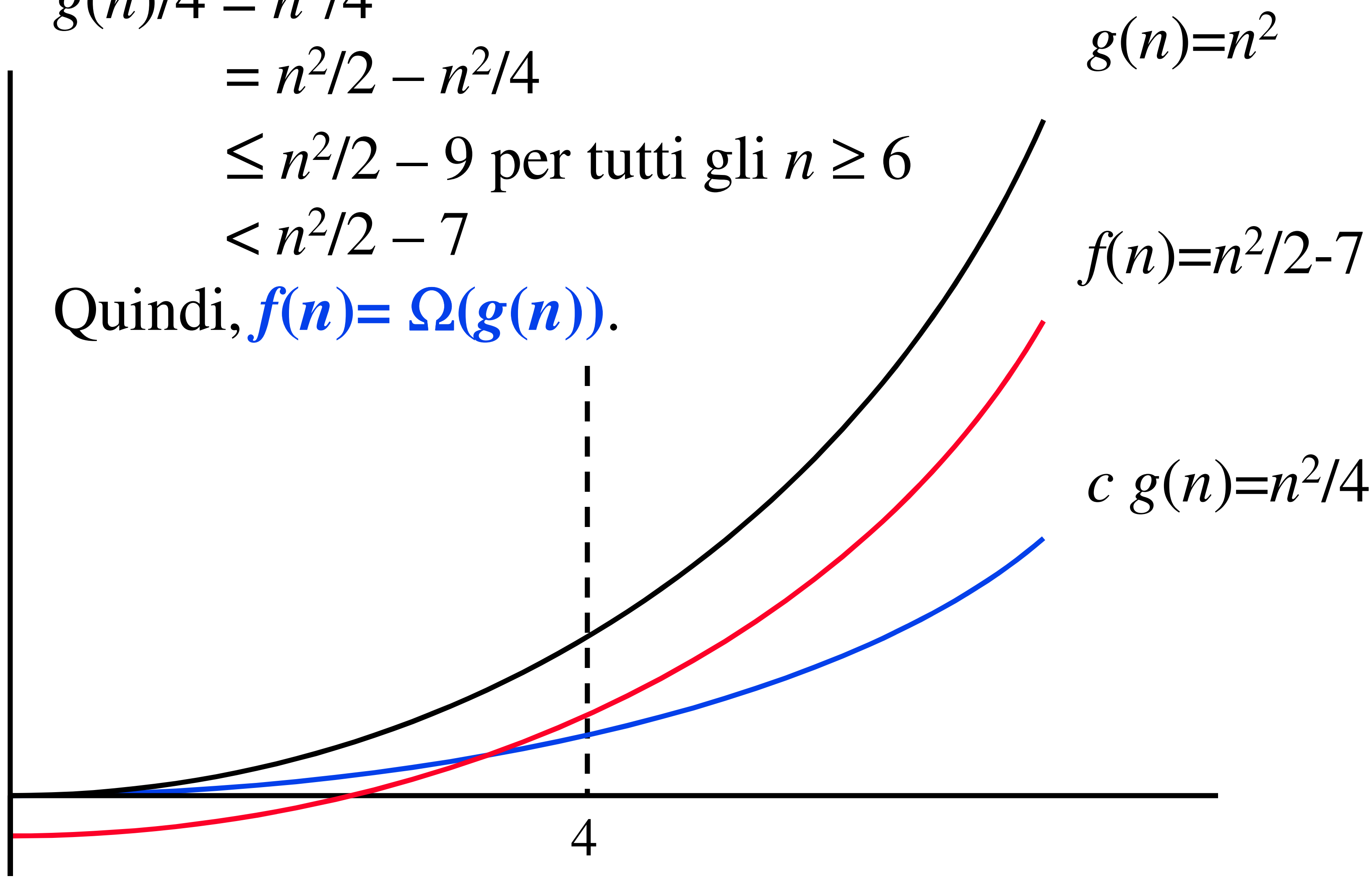
- $f(n) \geq c\ g(n)$ per tutti gli $n \geq n_0$
- $g(n)$ è detto un **limite inferiore asintotico** di $f(n)$.
- Scriviamo $\boldsymbol{f(n) = \Omega(g(n))}$
- Leggiamo ***f*(*n*) *è* Omega-grande di *g*(*n*)**.



Esempio di limite inferiore asintotico

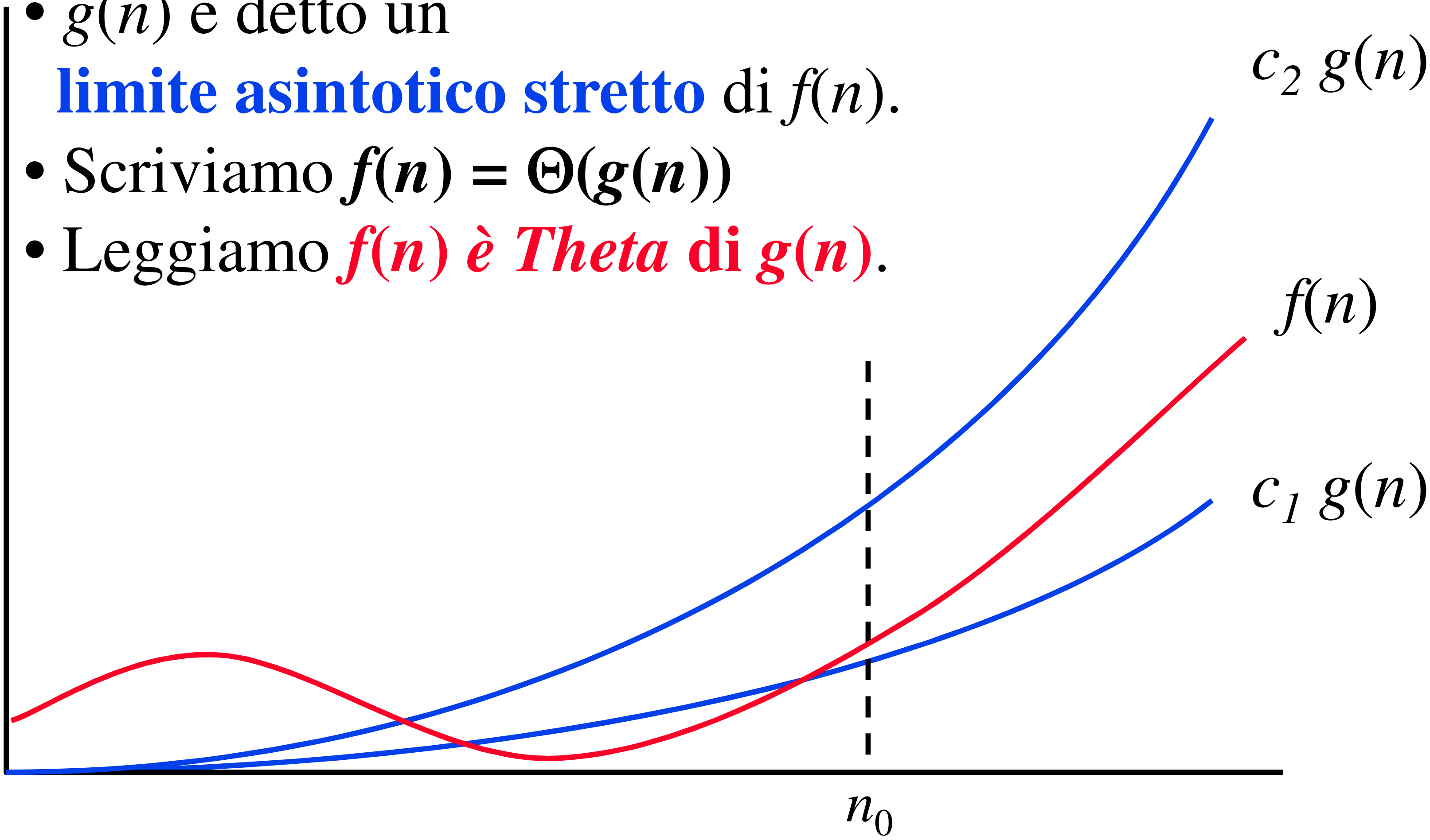
$$
\begin{aligned}
g(n)/4 &= n^2/4 \\
&= n^2/2 - n^2/4 \\
&\le n^2/2 - 9 \text{ per tutti gli } n \ge 6 \\
&< n^2/2 - 7
\end{aligned}
$$

Quindi, $\boldsymbol{f(n) = \Omega(g(n))}$.



Limite asintotico stretto

- $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$
- $g(n)$ è detto un **limite asintotico stretto** di $f(n)$.
- Scriviamo $\boldsymbol{f(n) = \Theta(g(n))}$
- Leggiamo ***f*(*n*) è *Theta* di *g*(*n*)**.



La ricerca dicotomica – 1

- **Per “cercare” un elemento in un vettore ordinato esiste un metodo detto ricerca binaria o dicotomica**
  - **Si confronta il valore val da ricercare con l’elemento centrale del vettore A[length/2]**
  - **Se val è minore dell’elemento mediano, si ripete la ricerca sulla metà sinistra del vettore, altrimenti si ricerca nella metà destra**

La ricerca dicotomica – 2

- **Esempio: ricerca del numero 23**

Si confronta 23 con 13

0	2	4	5	8	9	13	16	20	23	27	30	34	35

Ci si concentra sulla metà destra (da ind. 8 a ind. 14): si confronta 23 con 27

0	2	4	5	8	9	13	16	20	23	27	30	34	35

Ci si concentra sulla metà sinistra (da ind. 8 a ind. 10): si confronta 23 con 20

0	2	4	5	8	9	13	16	20	23	27	30	34	35

Ci si concentra sulla metà destra (da ind. 9 a ind. 9): trovato!!

0	2	4	5	8	9	13	16	20	23	27	30	34	35

Implementazione della ricerca dicotomica

```
int search(int val, int A[], int from, int to)
{
      int center=(from+to)/2;

      if (from > to) return –1;
      if (from==to) {
                if (A[from]==val) {return from;}
                return –1;} // si esegue solo se A[from]!=val

      //si esegue solo se (from<to)
      if (val<A[center]){ return search(val,A,from,center–1);}
      if (val>A[center]){ return search(val,A,center+1,to);}
      return center;

}
```

Gli algoritmi di ordinamento – 1

- L'ordinamento di una sequenza di informazioni consiste nel disporre le stesse informazioni in modo da rispettare una qualche relazione d'ordine; ad esempio, una relazione d'ordine "minore o uguale" dispone le informazioni in modo "non decrescente"
- L'ordinamento è un'operazione molto importante perché permette di ridurre notevolmente i tempi di ricerca di un'informazione, nell'ambito di una sequenza di informazioni
- Nel caso in cui tale sequenza risulta ordinata, secondo una qualche relazione d'ordine, è infatti possibile sfruttare la stessa relazione d'ordine per effettuare la ricerca

Gli algoritmi di ordinamento – 2

- Esistono due categorie di algoritmi di ordinamento: la classificazione è fatta in base alla complessità di calcolo e alla semplicità algoritmica
- La complessità di calcolo si riferisce al numero di operazioni necessarie all'ordinamento; tali operazioni sono essenzialmente confronti e scambi tra gli elementi dell'insieme da ordinare
- La semplicità algoritmica si riferisce alla lunghezza e alla comprensibilità del codice

Gli algoritmi di ordinamento – 3

- Algoritmi semplici di ordinamento

  Algoritmi che presentano complessità $\mathcal{O}(n^2)$, dove n è il numero di informazioni da ordinare: sono caratterizzati da poche e semplici istruzioni, dunque si realizzano con poche linee di codice

- Algoritmi evoluti di ordinamento

  Algoritmi che presentano complessità computazionale $\mathcal{O}(n \times \log_2 n)$: sono più complessi, fanno spesso uso di ricorsione; la convenienza del loro utilizzo si rileva quando il numero n di informazioni da ordinare è molto elevato

Bubblesort – 1

- La strategia Bubblesort (ordinamento a bolla) prevede il confronto dei primi due elementi di un array, e lo scambio, se il primo è maggiore del secondo
- Dopo il primo confronto, si effettua un confronto fra il secondo ed il terzo elemento (con eventuale scambio), fra il terzo ed il quarto, etc.
- Gli elementi “pesanti” (grandi) tendono a scendere verso il fondo del vettore, mentre quelli “leggeri” (più piccoli) salgono (come bolle) in superficie

Bubblesort – 2

- **Il confronto fra tutte le coppie di elementi adiacenti viene detto *passaggio***
  - **Se, durante il primo passaggio, è stato effettuato almeno uno scambio, occorre procedere ad un ulteriore passaggio**
  - **Ad ogni passaggio, almeno un elemento assume la sua posizione definitiva (l'elemento più grande del sottoinsieme attualmente disordinato)**
- **Devono essere effettuati al più n–1 passaggi**
- **Al k–esimo passaggio vengono effettuati n–k confronti (con eventuali scambi): almeno k–1 elementi sono già ordinati**
- **Complessivamente, vengono effettuati n×(n–1)/2 confronti**

  ⇒ **La complessità computazionale del Bubblesort è $\mathcal{O}(n^2)$**

Bubblesort – 3

```
#define FALSE 0
#define TRUE  1
#include <stdio.h>

void bubble_sort(list, list_size)
int list[], list_size;
{          int j, temp, sorted=FALSE;
         while (!sorted)
         {        sorted = TRUE;  /* assume che list sia ordinato */
                  for (j=0; j<list_size−1; j++)
                  {
                      if  (list[j]>list[j+1])
                      {   /* almeno un elemento non è in ordine  */
                          sorted = FALSE;
                          temp = list[j];
                          list[j] = list[j+1];
                          list[j+1] = temp;
                      }
                  }   /* fine del ciclo for */
         }   /* fine del ciclo while */
}
```

Nota

Nel caso migliore, quando il vettore è già ordinato, si effettua un solo passaggio, con n−1 confronti e nessuno scambio

⇨ La complessità scende a $\mathcal{O}$(n)