Algoritmi e Strutture Dati

Introduzione

Informazioni utili

- **T.H. Cormen, C.E. Leiserson, R.L Rivest, C. Stein "Introduzione agli algoritmi e strutture dati". McGraw-Hill**

- **Sito web con le slides del corso:**

  http://people.na.infn.it/~bene/ASD/

- **Orario di ricevimento: Mercoledì ore 11:00 – 13:00**

Algoritmo

Un **algoritmo** è una procedura ben definita per risolvere un problema: una sequenza di passi che, se eseguiti da un *esecutore*, portano alla ***soluzione del problema***.

La sequenza di passi che definisce un algoritmo deve essere ***descritta in modo finito***.

Alcune proprietà degli algoritmi

Non ambiguità: tutti i passi che definiscono l'algoritmo devono essere non ambigui e chiaramente comprensibili all'esecutore;

Generalità: la sequenza di passi da eseguire dipende esclusivamente dal problema generale da risolvere, non dai dati che ne definiscono un'istanza specifica;

Correttezza: un algoritmo è corretto se produce il risultato corretto a fronte di qualsiasi istanza del problema ricevuta in ingresso. Può essere stabilita, ad esempio, tramite:

- dimostrazione formale (matematica);
- ispezione informale;

Efficienza: misura delle risorse computazionali che esso impiega per risolvere un problema. Alcuni esempi sono:

- tempo di esecuzione;
- memoria impiegata;
- altre risorse: banda di comunicazione.

Complessità degli algoritmi

- **Analisi delle *prestazioni degli algoritmi***
- **Utilizzeremo un *Modello Computazionale* astratto di riferimento.**
- ***Tempo di esecuzione* degli algoritmi**
- ***Notazione asintotica***
- **Analisi del *Caso Migliore*, *Caso Peggiore* e del *Caso Medio***
- **Applicazione delle tecniche di analisi del tempo di esecuzione ad *algoritmi di ordinamento*.**

Analisi di un algoritmo

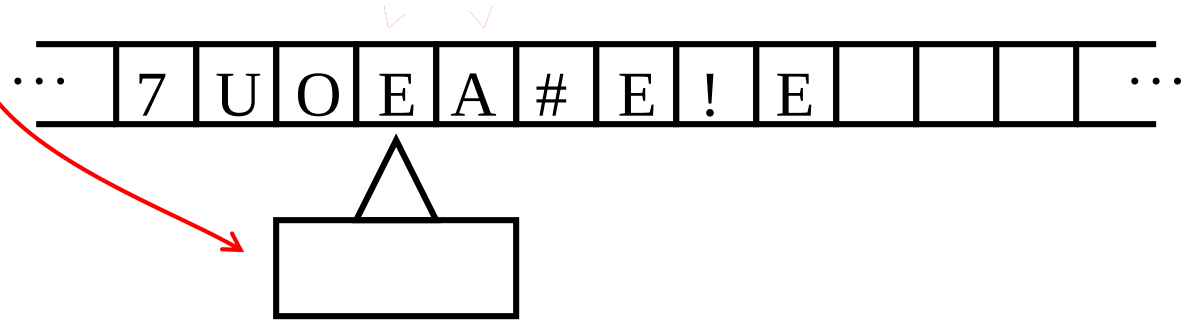
- **Correttezza**
  - **Dimostrazione formale (matematica)**
  - **Ispezione informale**
- **Utilizzo delle risorse**
  - **Tempo di esecuzione**
  - **Utilizzo della memoria**
  - **Altre risorse: banda di comunicazione**
- **Semplicità**
  - **Facile da capire, modificare e manutenere**

Tempo di esecuzione

- Il ***tempo di esecuzione*** di un ***programma*** può dipendere da vari fattori:
  - **Hardware su cui viene eseguito**
  - **Compilatore/Interprete utilizzato**
  - ***Tipo e dimensione dell'input***
  - **Altri fattori: casualità, …**
- Al fine di analizzare il ***tempo intrinseco impiegato*** da un algoritmo, procederemo a un'analisi più astratta, impiegando un ***modello computazionale***.

Un noto modello computazionale

- **Il modello della Macchina di Turing**
  - **Nastro di lunghezza infinita**
    - **In ogni *cella* può essere contenuta una quantità di informazione finita (un simbolo)**
  - **Una testina + un processore + programma**
    - **In 1 unità di tempo**
      - **Legge o scrive la cella di nastro corrente e**
      - **Si muove di 1 cella a sinistra, oppure di 1 cella a destra, oppure resta ferma**



Il modello computazionale RAM

Modello RAM (Random-Access Memory)

- **Memoria principale infinita**
  - **Ogni cella di memoria può contenere una quantità di dati finita.**
  - **Impiega lo stesso tempo per accedere a ogni cella di memoria.**
- **Singolo processore + programma**
  - **In 1 unità di tempo: operazioni di *lettura*, *passo* di *computazione elementare*, *scrittura*;**
  - **Passi di computazione: addizione, moltiplicazione, assegnamento, confronto, accesso a puntatore, …**

Il modello RAM è una semplificazione dei moderni computer.

Un problema di conteggio

- **Input**
  - **Un intero N dove $N \geq 1$.**
- **Output**
  - **Il numero di coppie ordinate (i, j) tali che i e j siano interi e $1 \leq i \leq j \leq N$.**

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

Algoritmo 4: analisi asintotica

```
int Count_0( int N)
1    sum = 0
2    for i =1 to N
3        for j =1 to N
4          if i <= j then
5            sum = sum+1
6    return sum
```

1

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

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

$2\sum_{i=1}^{N} N$

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

1

Ma notate che:

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

Algoritmo 4: analisi asintotica

```
int Count_0( int N)
1    sum = 0
2    for i =1 to N
3        for j =1 to N
4          if i <= j then
5             sum = sum+1
6    return sum
```

1

$2(N+1)$

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

$2\sum_{i=1}^{N} N$

$\frac{N(N+1)}{2}$

1

Il tempo di esecuzione è $T(N) = 9/2\ N^2 + 9/2\ N + 4$

Algoritmo 1

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

1

$2N+2$

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

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

1

Il tempo di esecuzione è

$$2+2N+2+2\sum_{i=1}^{N}(N+2-i)+2\sum_{i=1}^{N}(N+1-i)=2N^2+6N+4$$

Algoritmo 2

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

1

$2N+2$

$4N$

1

Il tempo di esecuzione è $6N+4$

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
2     return sum
```

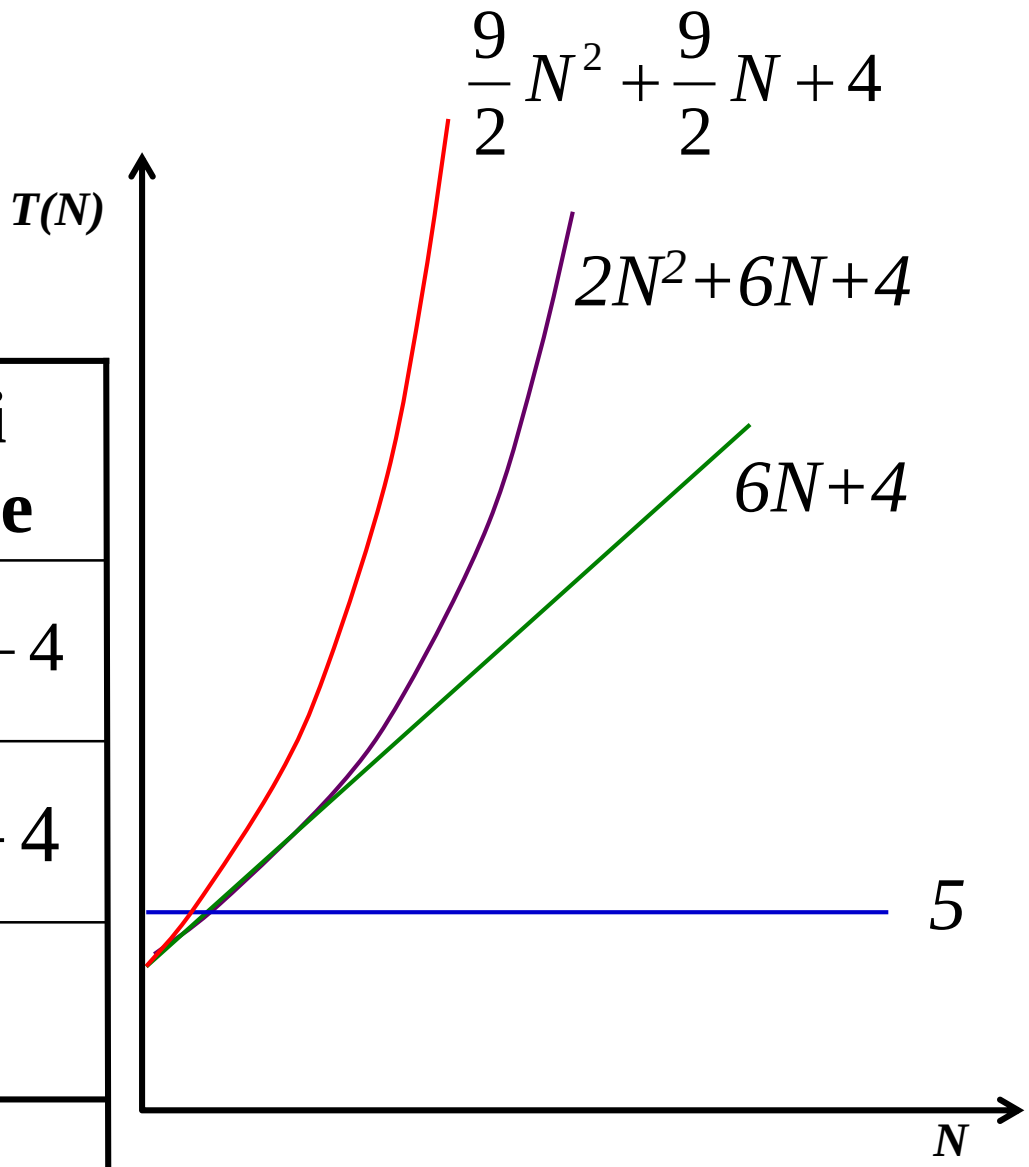
4

1

Il tempo di esecuzione è 5 unità di tempo

Riassunto dei tempi di esecuzione

Algoritmo	Tempo di Esecuzione
Algoritmo 0	$\frac{9}{2}N^2+\frac{9}{2}N+4$
Algoritmo1	$2N^2+6N+4$
Algoritmo 2	**$6N+4$**
Algoritmo 3	**5**



Ordine dei tempi di esecuzione

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

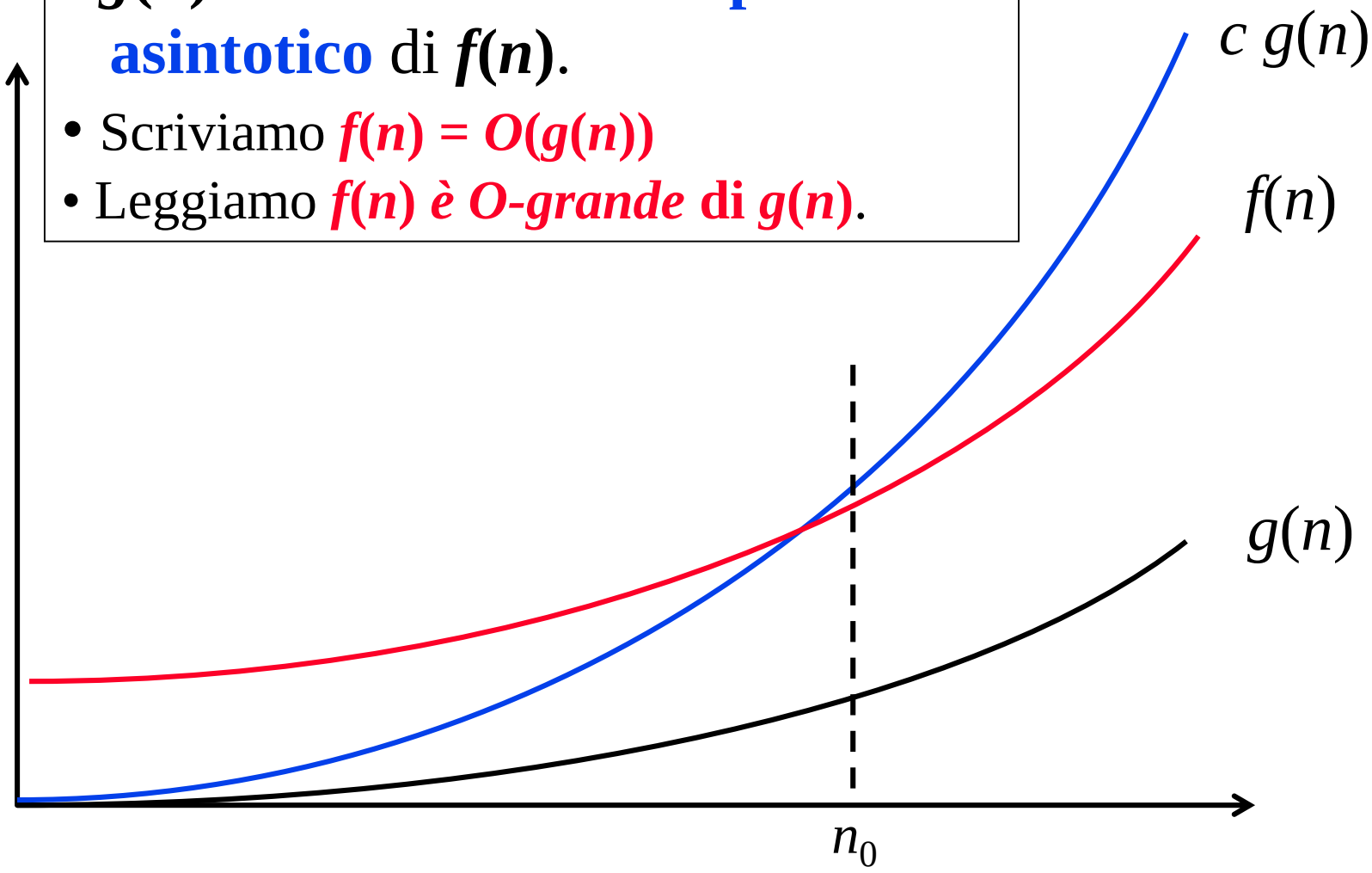
	1.000	10.000	100.000	1.000.000	10.000.000
N	1 μs	10 μs	100 μs	1 ms	10 ms
20N	20 μs	200 μs	2 ms	20 ms	200 ms
N Log N	9.96 μs	132 μs	1.66 ms	19.9 ms	232 ms
20N Log N	199 μs	2.7 ms	33 ms	398 ms	4.6 sec
N^2	1 ms	100 ms	10 sec	17 min	1.2 giorni
$20N^2$	20 ms	2 sec	3.3 min	5.6 ore	23 giorni
N^3	1 sec	17 min	12 gior.	32 anni	32 millenni

Riassunto dei tempi di esecuzione

Algoritmo	**Tempo di Esecuzione**	**Ordine del Tempo di Esecuzione**
Algoritmo 0	$\frac{9}{2}N^2+\frac{9}{2}N+4$	N^2
Algoritmo 1	$2N^2+6N+4$	N^2
Algoritmo 2	$6N+4$	N
Algoritmo 3	**5**	**Costante**

Limite superiore asintotico

- $\exists c>0, n_0>0 \ \forall n \geq n_0. \quad f(n) \leq c\, g(n)$
- $g(n)$ 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

$4\,g(n) = 4n^2$

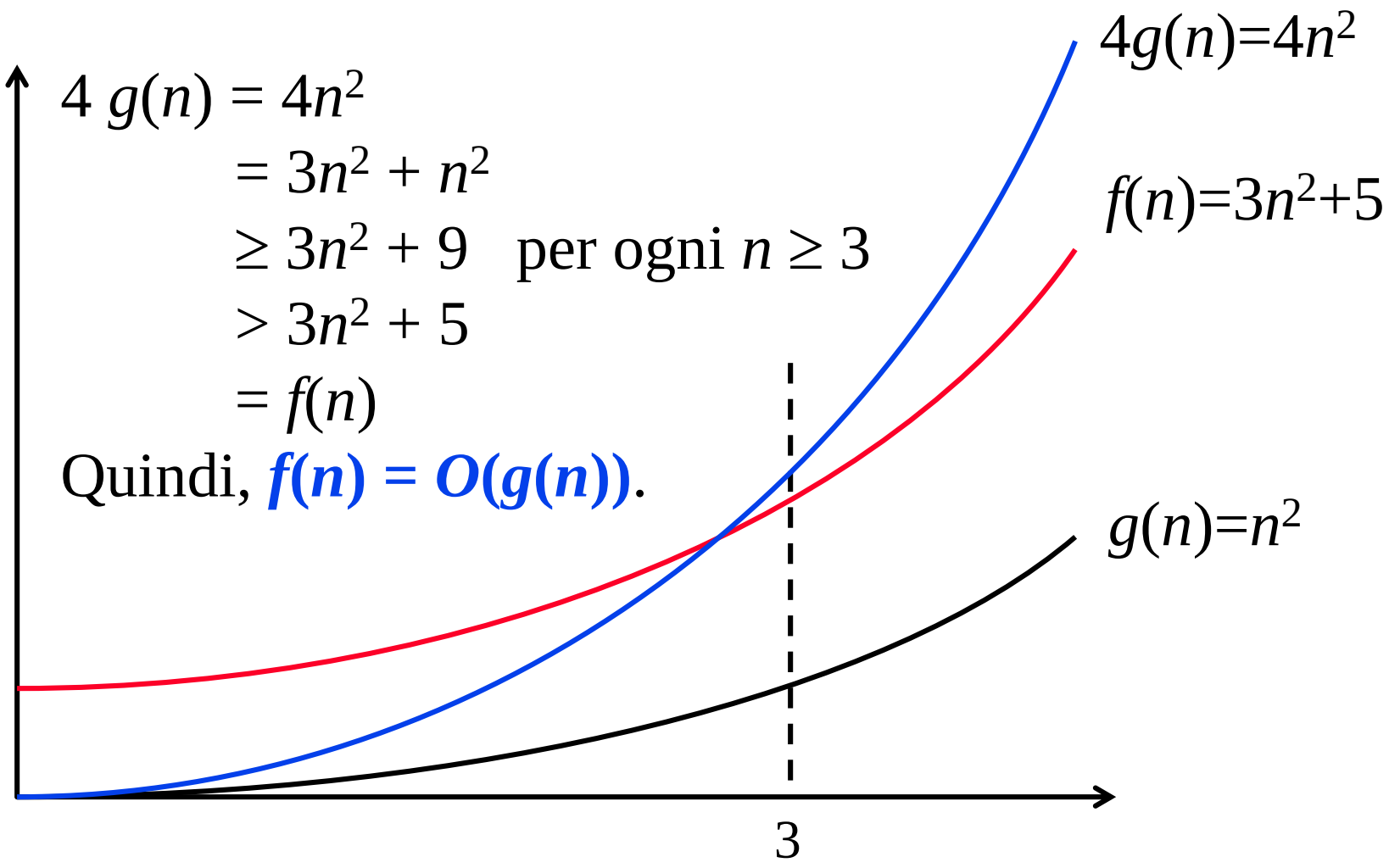
$= 3n^2 + n^2$

$\geq 3n^2 + 9$ per ogni $n \geq 3$

$> 3n^2 + 5$

$= f(n)$

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



Esercizio sulla notazione O

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

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

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

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 userà:**
    - $3n^2+2n+5 = O(3n^2+2n+5)$
    - $3n^2+2n+5 = O(n^2+n)$
    - $3n^2+2n+5 = O(3n^2)$

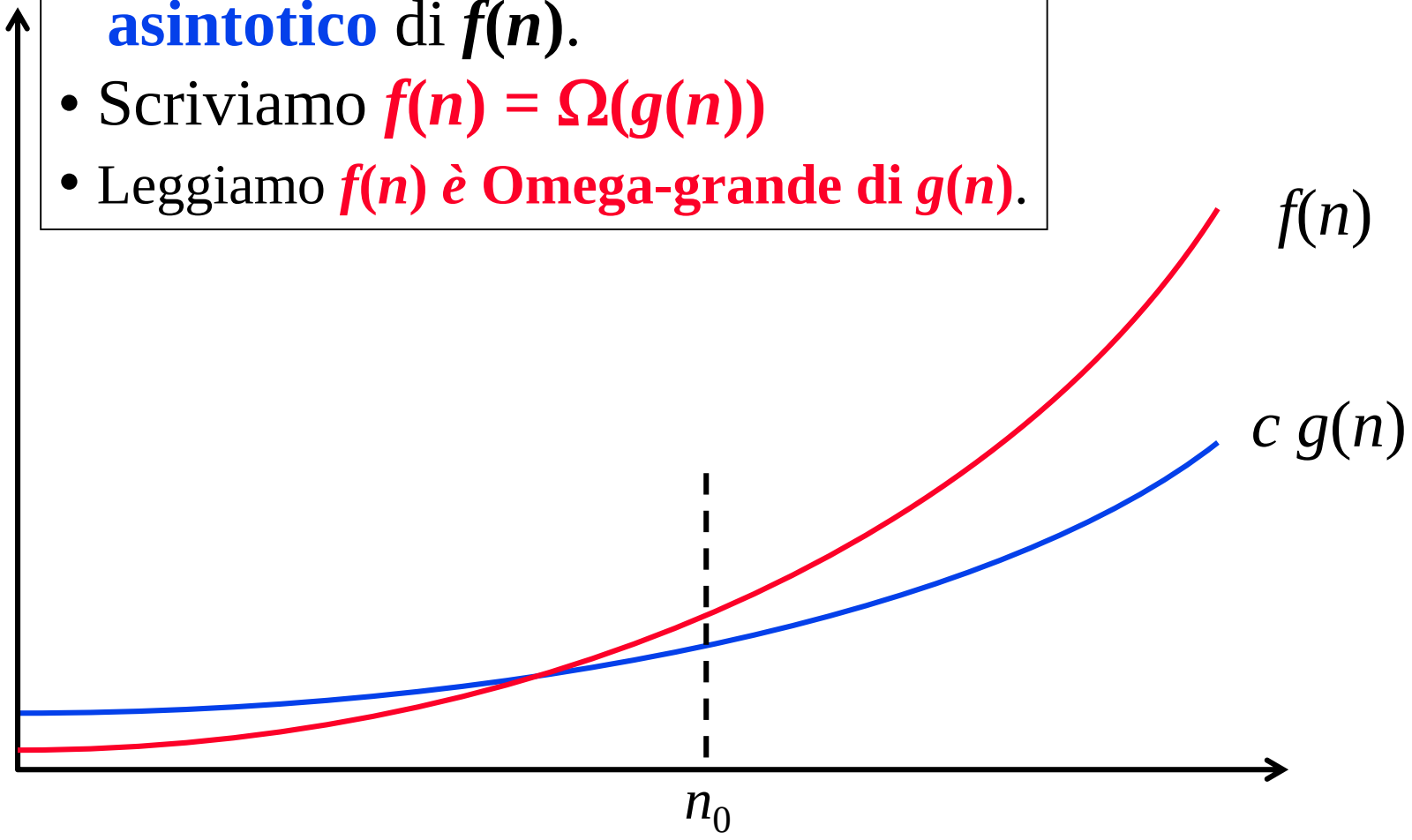
Esercizi sulla notazione O

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

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

Limite inferiore asintotico

- $\exists c>0, n_0>0 \ \forall n \geq n_0. \quad f(n) \geq c\, g(n)$
- $g(n)$ è detto un **limite inferiore asintotico** di $f(n)$.
- Scriviamo $f(n) = \Omega(g(n))$
- Leggiamo **$f(n)$ è Omega-grande di $g(n)$**.



Esempio di limite inferiore asintotico

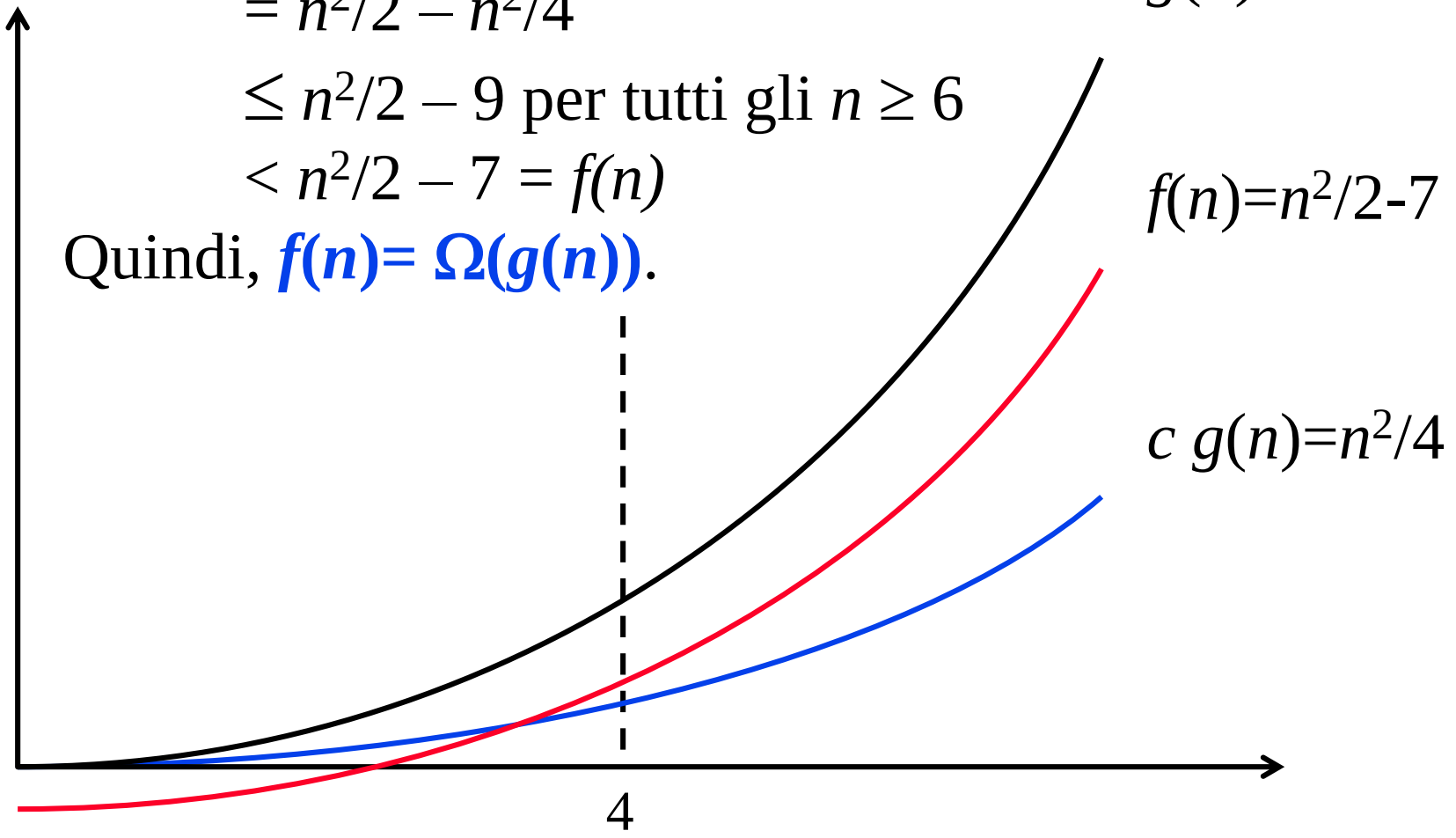
$g(n)/4 = n^2/4$

$= n^2/2 - n^2/4$

$\leq n^2/2 - 9$ per tutti gli $n \geq 6$

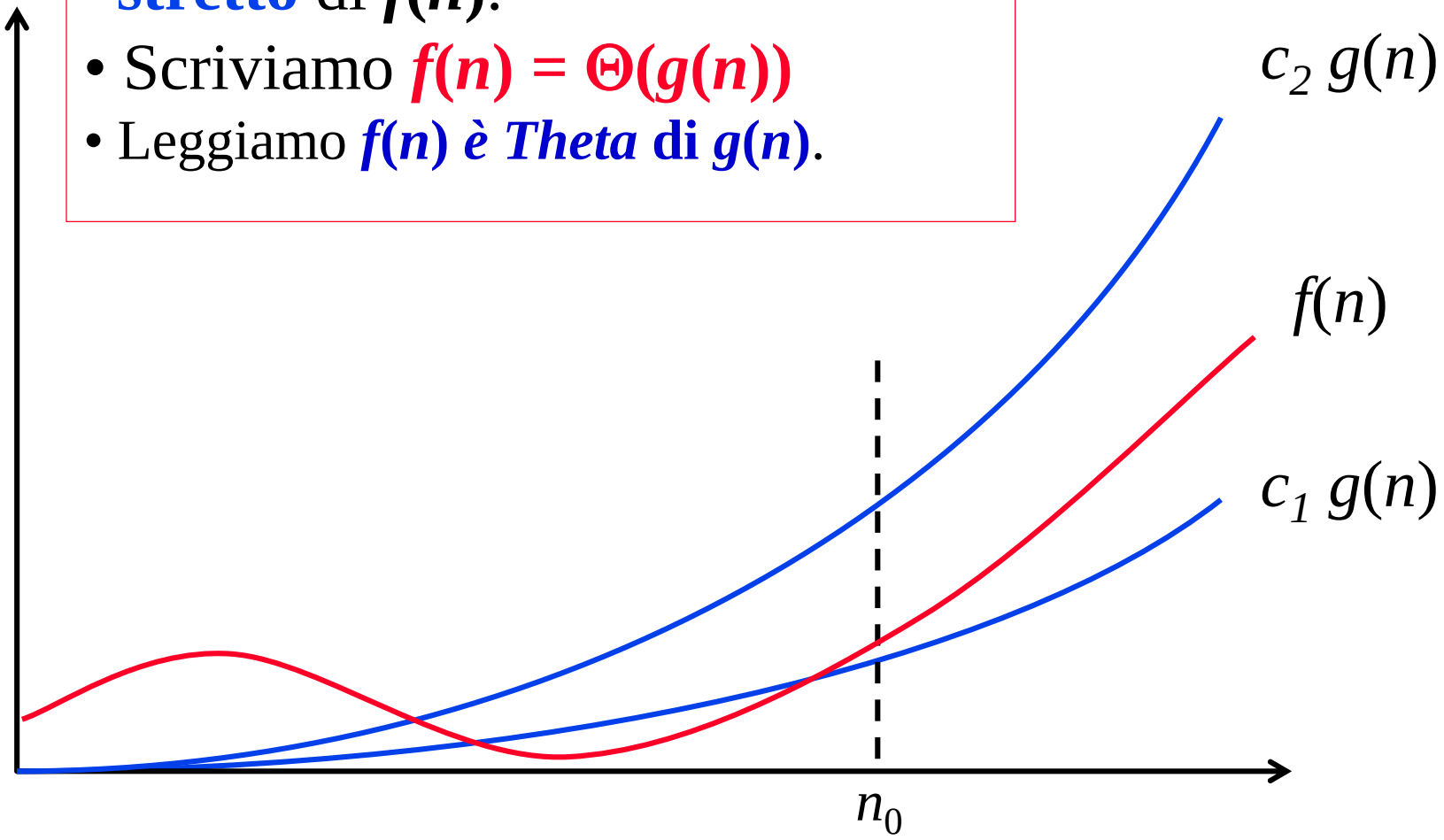
$< n^2/2 - 7 = f(n)$

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



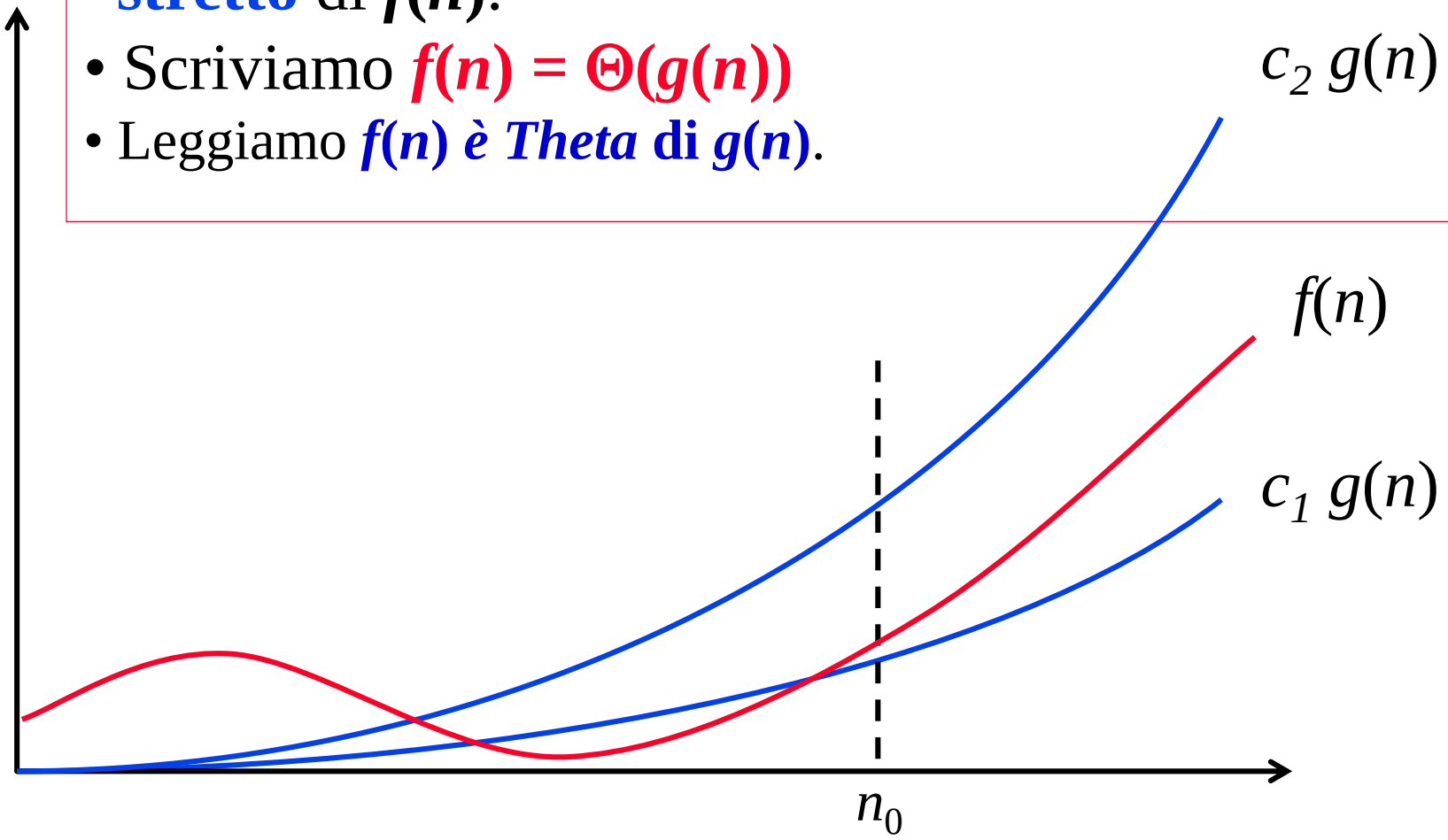
Limite asintotico stretto

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



Limite asintotico stretto

- $\exists c_1, c_2 > 0, n_0 > 0 \ \forall n \geq n_0 .\ c_1\, g(n) \leq f(n) \leq c_2\, g(n)$
- $g(n)$ è detto un **limite asintotico stretto** di $f(n)$.
- Scriviamo $f(n) = \Theta(g(n))$
- Leggiamo ***f(n) è Theta di g(n)***.



Riassunto della notazione asintotica

- ***O*: *O-grande*: limite superiore asintotico**
- **Ω: *Omega-grande*: limite inferiore asintotico**
- **Θ: *Theta*: limite asintotico stretto**

- **Usiamo la *notazione asintotica* per dare un limite ad una funzione (*f*(*n*)), a meno di un fattore costante (*c*).**

Teoremi sulla notazione asintotica

***Teoremi*:**

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

2. **Se** $f_1(n)=O(f_2(n))$ **e** $f_2(n)=O(f_3(n))$, **allora** $f_1(n)=O(f_3(n))$

3. **Se** $f_1(n)=\Omega(f_2(n))$ **e** $f_2(n)=\Omega(f_3(n))$, **allora** $f_1(n)=\Omega(f_3(n))$

4. **Se** $f_1(n)=\Theta(f_2(n))$ **e** $f_2(n)=\Theta(f_3(n))$, **allora** $f_1(n)=\Theta(f_3(n))$

5. **Se** $f_1(n) = O(g_1(n))$ **e** $f_2(n) = O(g_2(n))$, **allora**

$$O(f_1(n) + f_2(n)) = O(\max\{g_1(n), g_2(n)\})$$

6. **Se** $f(n)$ **è un** *polinomio* **di grado** d, **allora** $f(n) = \Theta(n^d)$

Teoremi sulla notazione asintotica

Proprietà:

Se $\lim_{n\to\infty} f(n)/g(n) = 0$ allora $f(n) = O(g(n))$

Se $\lim_{n\to\infty} f(n)/g(n) = k > 0$ allora $f(n) = O(g(n))$

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

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

Se $\lim_{n\to\infty} f(n)/g(n) \to \infty$ allora $f(n) = \Omega(g(n))$

Tempi di esecuzione asintotici

Algoritmo	**Tempo di Esecuzione**	**Limite asintotico**
Algoritmo 1	$2N^2+6N+4$	$O(N^2)$
Algoritmo 2	$6N+4$	$O(N)$
Algoritmo 3	**5**	$O(1)$
Algoritmo 4	$4N^2+5N+4$	$O(N^2)$

Somma Massima di una sottosequenza contigua

- **Input**
  - **Un intero N dove $N \geq 1$.**
  - **Una sequenza $(a_1, a_2, \ldots, a_N)$ di N interi.**
- **Output**
  - **Un intero S tale che $S = \sum_{k=i}^{j} a_k$ dove $1 \leq i, j \leq N$ e S è il più grande possibile.**
  - **(tutti gli elementi nella sommatoria devono essere contigui nella sequenza in input).**
    - Esempio:
      - N=9, (2,-4,**8,3,-5,4,6**,-7,2)
      - Output = **8+3-5+4+6** = **16**

Algoritmo 1

```
int Max_seq_sum_1(int N, array a[])
  maxsum = 0                                          O(1)
  for i=1 to N                                        O(N)
    for j=i to N                                      O(N^2)
      sum = 0
      for k=i to j                                    O(N^3)
        sum = sum + a[k]
      maxsum = max(maxsum,sum)
  return maxsum
```

Tempo di esecuzione $O(N^3)$

Algoritmo 2

- È facile osservare che l'algoritmo precedente effettua spesso le **stesse operazioni ripetutamente**.
- Poichè

$$\sum_{k=i}^{j+1} a_k = a_{j+1} + \sum_{k=i}^{j} a_k$$

è possibile ottenere il valore di **sum** per la sequenza da **i** a **j+1** in tempo costante, sommando **A[j+1]** al valore di **sum** già calcolato all'itarazione precedente per la sequenza da **i** a **j**.

- A tal fine, è sufficiente mantenere inalterato il valore di **sum** tra le iterazioni che individuano sottosequenze che partono dallo stesso valore **i** e riazzerare **sum** solo quando **i** viene incrementato.

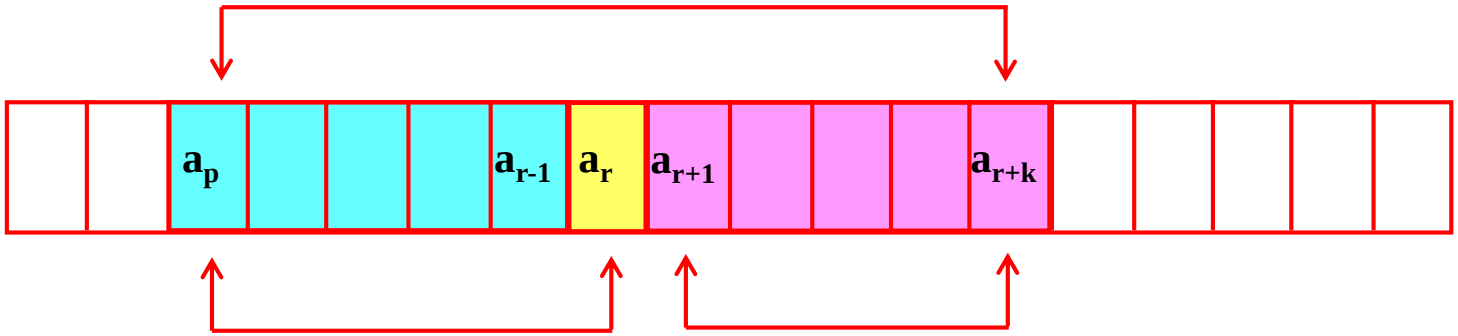
Algoritmo 2

```
int Max_seq_sum_2(int N, array a[])
   maxsum = 0                                          O(1)
   for i=1 to N                                        O(N)
      sum = 0
      for j=i to N                                     O(N^2)
         sum = sum + a[j]
         maxsum = max(maxsum,sum)
   return maxsum
```

Tempo di esecuzione $O(N^2)$

Esiste un algoritmo che risolve il problema in tempo $O(N)$

Algoritmo 3: intuizione



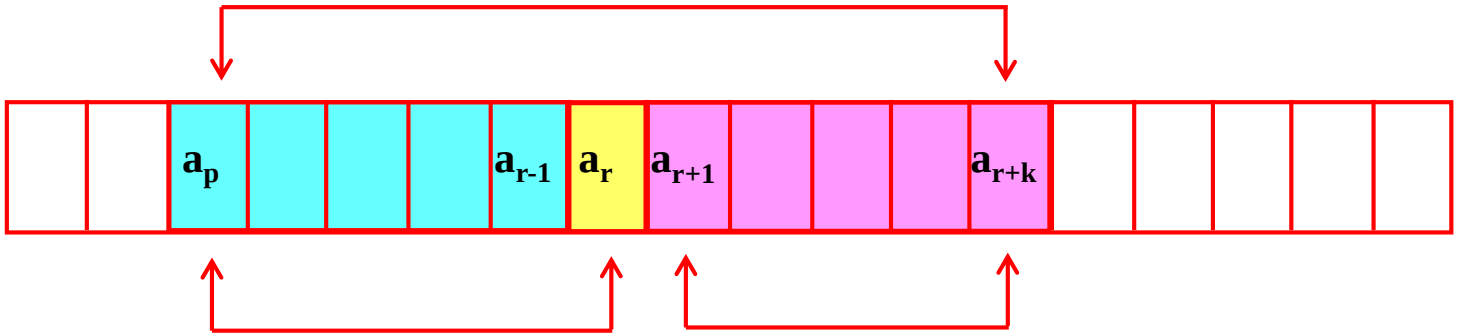
1. **Se $a_p+\ldots+a_r \geq 0$ allora**

$$a_p+\ldots+a_{r+k} \geq a_{r+1}+\ldots+a_{r+k} \quad \forall\ k\geq 1$$

2. **Se $a_p+\ldots+a_{r-1} > 0$ ma $a_p+\ldots+a_r < 0$ allora**

$$a_p+\ldots+a_{r+k} \leq a_{r+1}+\ldots+a_{r+k} \quad \forall\ k\geq 1$$

Algoritmo 3: intuizione



1. **Se $a_p+\ldots+a_r \geq 0$ allora**

$$a_p+\ldots+a_{r+k} \geq a_{r+1}+\ldots+a_{r+k} \quad \forall\, k\geq 1$$

2. **Se $a_p+\ldots+a_{r-1} > 0$ ma $a_p+\ldots+a_r < 0$ allora**

$$a_p+\ldots+a_{r+k} \leq a_{r+1}+\ldots+a_{r+k} \quad \forall\, k\geq 1$$

Nel caso **2**, ogni sottosequenza di **A** che inizia tra **p** e **r** e che termina oltre **r** avrà una ***somma inferiore alla sua sottosequenza*** che parte da **r+1**.

È dunque possibile ***ignorare tutte queste sottosequenze*** e considerare solo quelle che iniziano dall'indice **r+1**.

Algoritmo 3

```
int Max_seq_sum_3(int N, array a[])
  maxsum = 0                                        O(1)
  sum = 0
  for j=1 to N                                      O(N)
     if (sum + a[j] > 0) then
          sum = sum + a[j]
     else
          sum = 0
     maxsum = max(maxsum,sum)
  return maxsum
```

Tempo di esecuzione $O(N)$

Ordinamento di una sequenza

- **Input : una sequenza di numeri.**
- **Output : una permutazione (riordinamento) tale che tra ogni 2 elementi adiacenti nella sequenza valga "qualche" relazione di ordinamento (ad es. ≤).**
- ***Insert Sort***
  - **È efficiente solo per piccole sequenze di numeri;**
  - **Algoritmo di ordinamento sul posto.**

1) **La sequenza viene scandita dal dal primo elemento; l'indice *i*, *inizialmente* assegnato al primo elemento, indica l'elemento corrente;**
2) **Si considera la parte a sinistra di *i* (compreso) già ordinata;**
3) **Si seleziona il primo elemento successivo ad *i* nella sottosequenza non-ordinata assegnando *j = i+1*;**
4) **Si cerca il posto giusto per l'elemento *j* nella sottosequenza ordinata.**
5) **Si incrementa *i*, si torna al passo 3) se la sequenza non è terminata;**

Insert Sort

Algoritmo :

- ***A*[1..*n*]** : sequenza numeri di input
- ***Key*** **:** valore corrente da inserire nell'ordinamento

```
1  for j = 2 to Lenght(A)
2     do Key = A[j]
      /* Scelta del j-esimo elemento da ordinare */
3     i = j-1   /* A[1…i] è la porzione ordinata */
4     while i > 0 and A[i] > Key do
5        A[i+1] = A[i]
6        i=i-1
7     A[i+1] = Key
```

Analisi di Insert Sort

		Numero Esecuzioni	Costo esecuzione singola
1	for j = 2 to Lenght(A)	n	c_1
2	do Key = A[j]	$n-1$	c_2
	/* Commento */	$n-1$	0
3	i = j-1	$n-1$	c_3
4	while i>0 and A[i] > Key		c_4
5	do A[i+1] = A[i]		c_5
6	i=i-1		c_6
7	A[i+1] = Key	$n-1$	c_7

Analisi di Insert Sort

		Numero Esecuzioni	Costo esecuzione singola
1	for j = 2 to Lenght(A)	n	c_1
2	do Key = A[j]	$n-1$	c_2
	/* Commento */	$n-1$	0
3	i = j-1	$n-1$	c_3
4	while i>0 and A[i] > Key	$\sum_{j=2}^{n} t_j$	c_4
5	do A[i+1] = A[i]	$\sum_{j=2}^{n} (t_j - 1)$	c_5
6	i=i-1	$\sum_{j=2}^{n} (t_j - 1)$	c_6
7	A[i+1] = Key	$n-1$	c_7

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Analisi di Insert Sort: Caso migliore

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Il caso migliore si ha quando l'array è già ordinato:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_7(n-1)$$

Inoltre, in questo caso t_j è **1**, quindi:

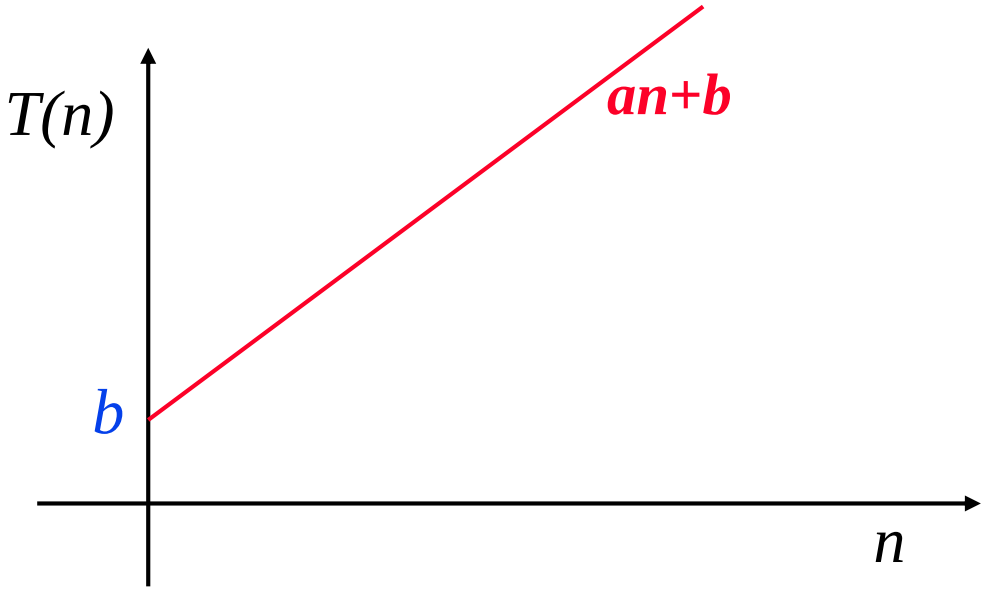
$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

T(n) = an+b

Analisi di Insert Sort: Caso migliore

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

T(n) = an+b



Analisi di Insert Sort: Caso peggiore

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Il caso peggiore si ha quando l'array è in ordine inverso.
In questo caso t_j è j (perché?)

$$\sum_{j=2}^{n} t_j = \sum_{j=1}^{n} t_j - 1 = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} t_j - \sum_{j=2}^{n} 1 = \frac{n(n+1)}{2} - 1 - (n-1) = \frac{n(n-1)}{2}$$

Quindi:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) +$$
$$+ c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1)$$

Analisi di Insert Sort: Caso peggiore

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1)$$

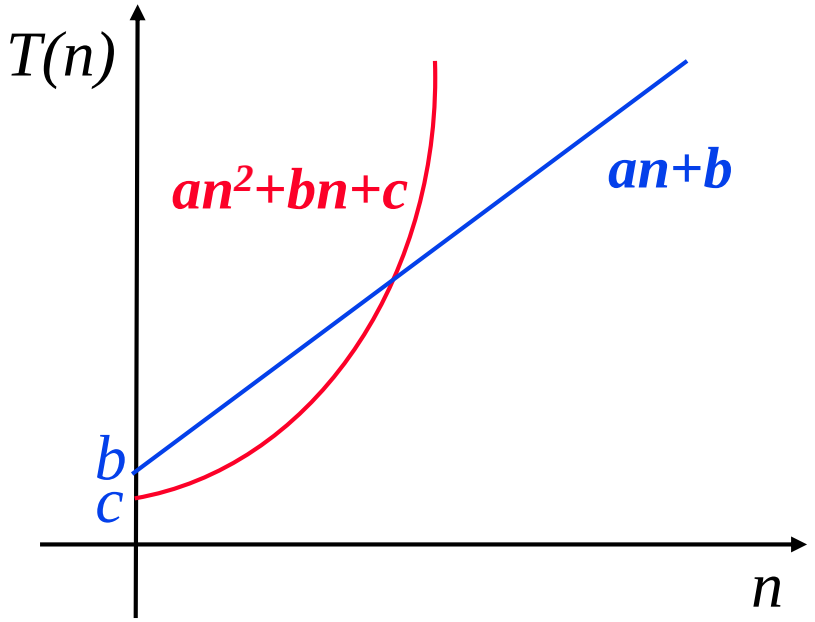
$$T(n) = \left(\frac{c_4 + c_5 + c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4 - c_5 - c_6}{2} + c_7\right)n - (c_2 + c_3 + c_4 + c_7)$$

$$T(n) = an^2 + bn + c$$

Analisi di Insert Sort: Caso peggiore

$$T(n) = \left(\frac{c_4 + c_5 + c_6}{2}\right) n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4 - c_5 - c_6}{2} + c_7\right) n - (c_2 + c_3 + c_4 + c_7)$$

$$T(n) = an^2 + bn + c$$



Analisi di Insert Sort: Caso medio

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Il ***caso medio*** è il valore medio del tempo di esecuzione.

Supponiamo di scegliere una ***sequenza casuale*** e che tutte le sequenze abbiano uguale probabilità di essere scelte.

In media, ***metà degli elementi*** ordinati saranno ***maggiori*** dell'elemento che dobbiamo sistemare.

In media ***controlliamo metà del sottoarray*** ad ogni ciclo **`while`**.

Quindi t_j è circa $j/2$.

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} \frac{j}{2} = \frac{1}{2}\left[\left(\sum_{j=1}^{n} j\right) - 1\right] = \frac{n^2 + n - 2}{4}$$

$$\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} \left(\frac{j}{2} - 1\right) = \frac{n^2 - 3n + 2}{4}$$

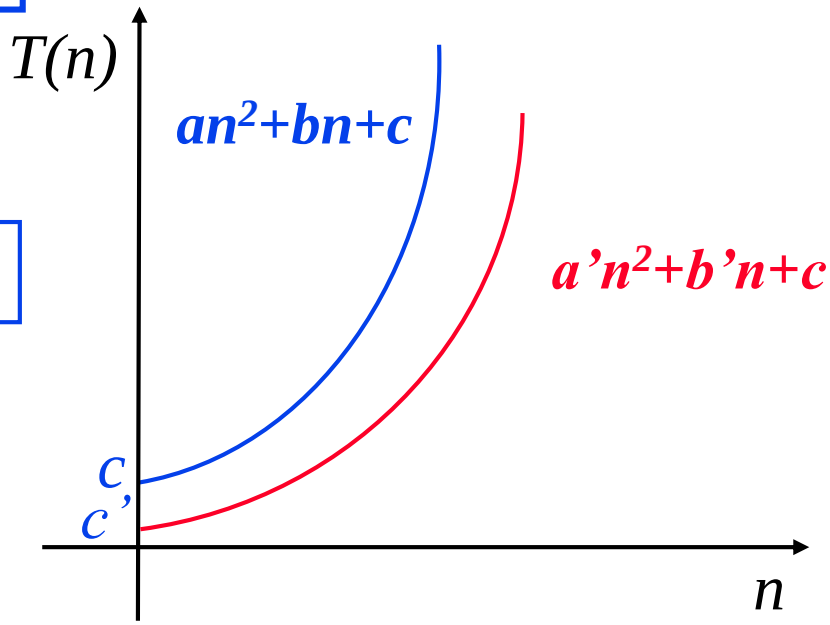
Analisi di Insert Sort: Caso medio

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} \frac{j}{2} = \frac{1}{2}\left(\sum_{j=1}^{n} j - 1 \right) = \frac{n^2 + n - 2}{4}$$

$$\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} \left(\frac{j}{2} - 1 \right) = \frac{n^2 - 3n + 2}{4}$$

$$T(n) = a'n^2 + b'n + c'$$



Analisi del Caso Migliore e Caso Peggiore

- **Analisi del Caso Migliore**
  - **Ω-grande, limite inferiore, del tempo di esecuzione per un qualunque *input di dimensione N*.**
- **Analisi del Caso Peggiore**
  - ***O*-grande, limite superiore, del tempo di esecuzione per un qualunque *input di dimensione N*.**

Analisi del Caso Medio

- **Analisi del Caso Medio**
  - **Alcuni algoritmi sono efficienti in pratica.**
  - **L'analisi è in genere molto più difficile.**
  - **Bisogna generalmente assumere che tutti gli input siano ugualmente probabili.**
  - **A volte non è ovvio quale sia la media.**

Stima del limite asintotico superiore

- **Nei prossimi lucidi vedremo un semplice metodo per *stimare il limite asintotico superiore O(.)* del tempo di esecuzione di *algoritmo iterativi*.**
  - **Stabilire il limite superiore per le operazioni elementari**
  - **Stabilire il limite superiore per le strutture di controllo**

- **Ci da un limite superiore che funge da stima, ma *non garantisce* di trovare la *funzione esatta* del *tempo di esecuzione*. La stima può essere a volte grossolana.**

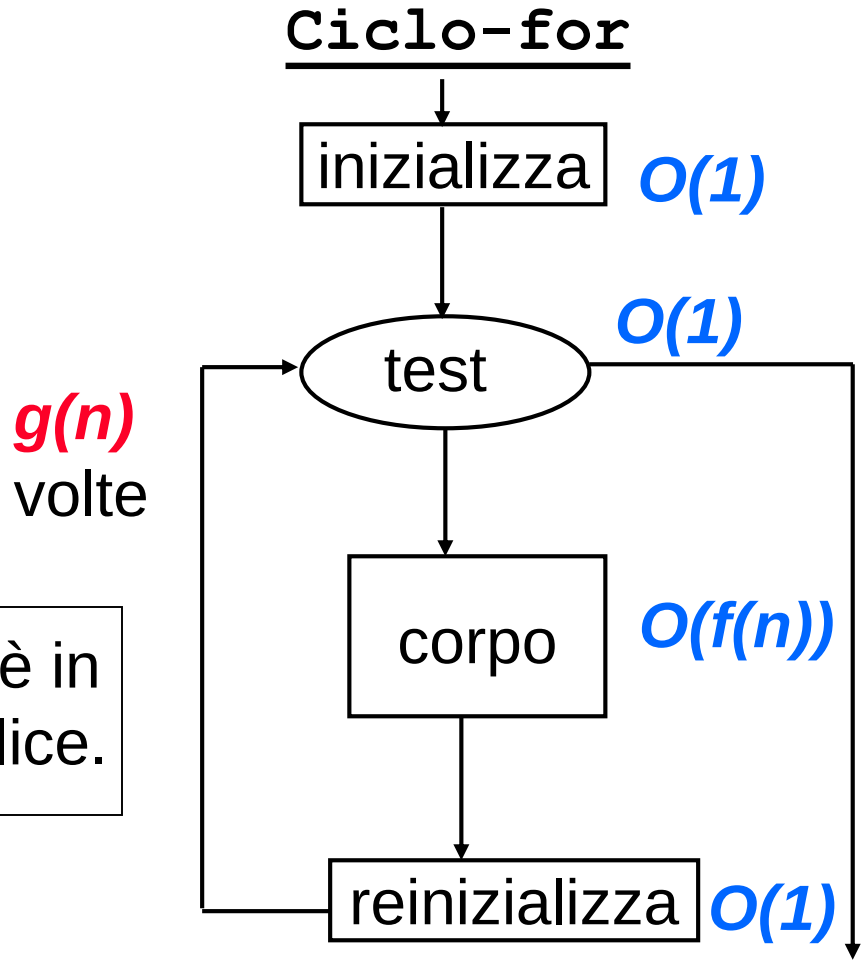
Tempo di esecuzione: operazioni semplici

Operazioni Semplici

- *operazioni aritmetiche (+, *,…)*
- *operazioni logiche(&&, ||,….)*
- *confronti (≤ ,≥ , = ,…)*
- *assegnamenti* **(`a = b`) senza chiamate di funzione**
- *operazioni di lettura (`read`)*
- *operaioni di controllo* **(`break`, `continue`, `return`)**

$$T(n) = \Theta(1) \Rightarrow T(n) = O(1)$$

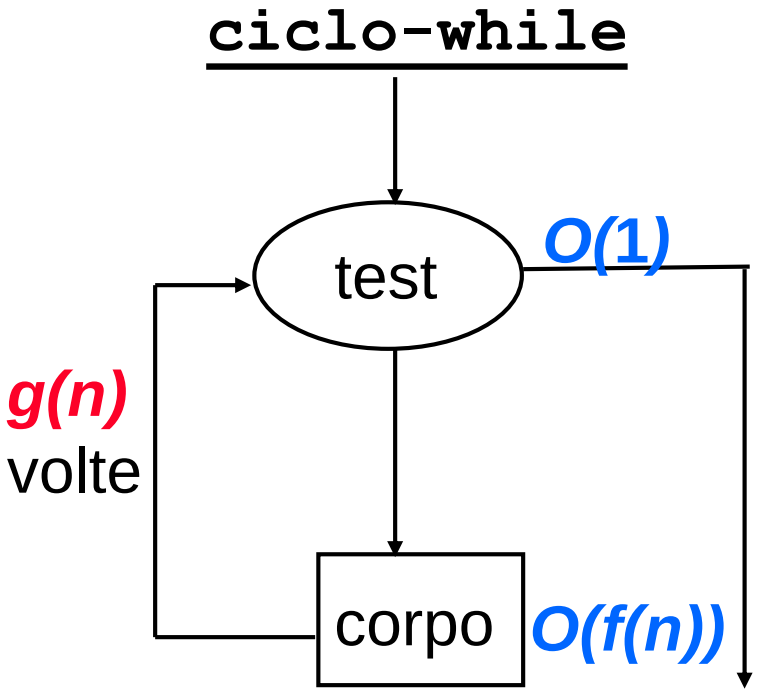
Tempo di esecuzione: ciclo for



stabilire ***g(n)*** è in genere semplice.

$$T(n) = O(g(n) \times f(n))$$

Tempo di esecuzione: ciclo while



Bisogna stabilire un limite per il numero di iterazioni del ciclo, ***g(n)***.

Può essere necessaria una prova induttiva per ***g(n)***.

$$T(n) = O(g(n) \times f(n))$$

Ciclo while: esempio

Ricerca dell'elemento ***x*** all'interno di un array ***A[1…n]***:

```
i = 1                                   (1)
while (x ≠ A[i] && i≤n)                 (2)
     i = i+1                            (3)
```

(1) ***O*(1)**

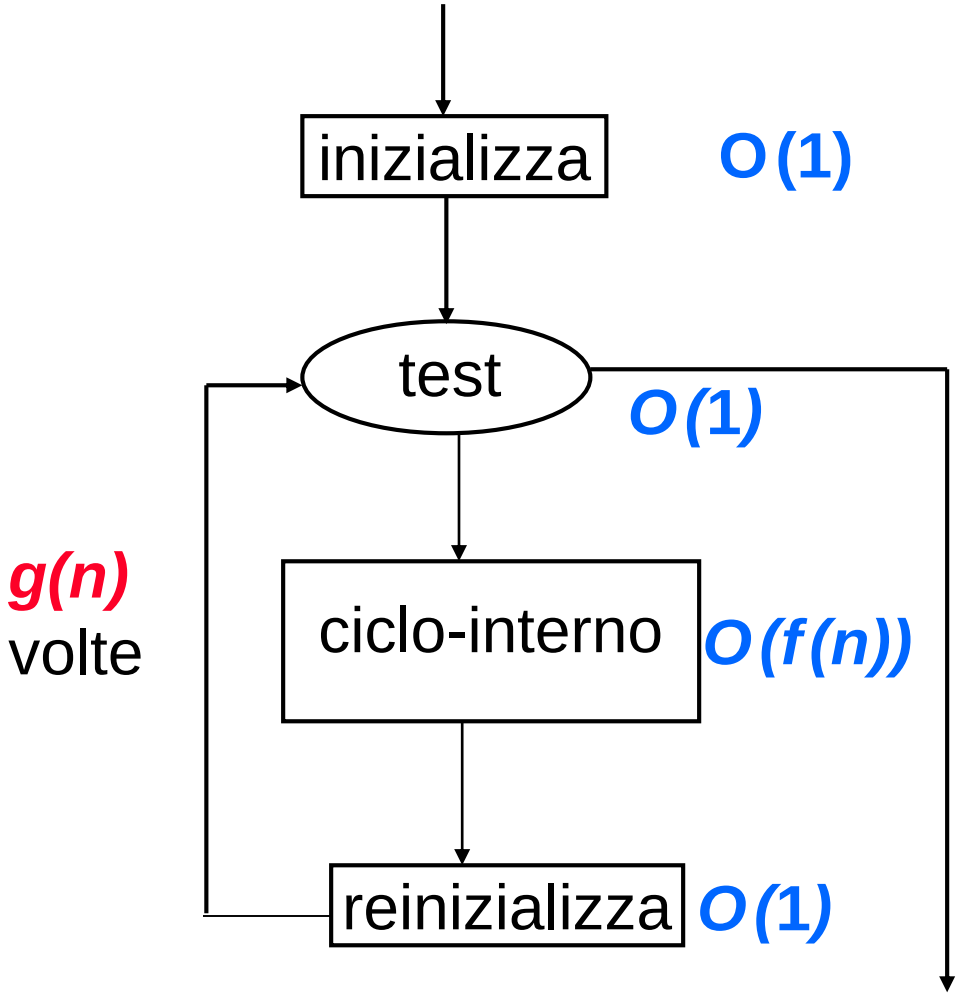
test in (2) ***O*(1)**

(3) ***O*(1)**

iterazioni massimo g(n) = ***n***

O(ciclo-while) = O(1) + n O(1) = O(n)

Tempo di esecuzione: cicli innestati



$$T(n) = O(g(n) \times f(n))$$

Cicli annidati: esempio

```
for i = 1 to n
    for j = 1 to n
        k = i + j
```

$$\left.\begin{array}{l}\\ \\ \\ \end{array}\right\} = O(n) \Bigg\} = O(n^2)$$

$$T(n) = O(n \times n) = O(n^2)$$

Cicli annidati: esempio

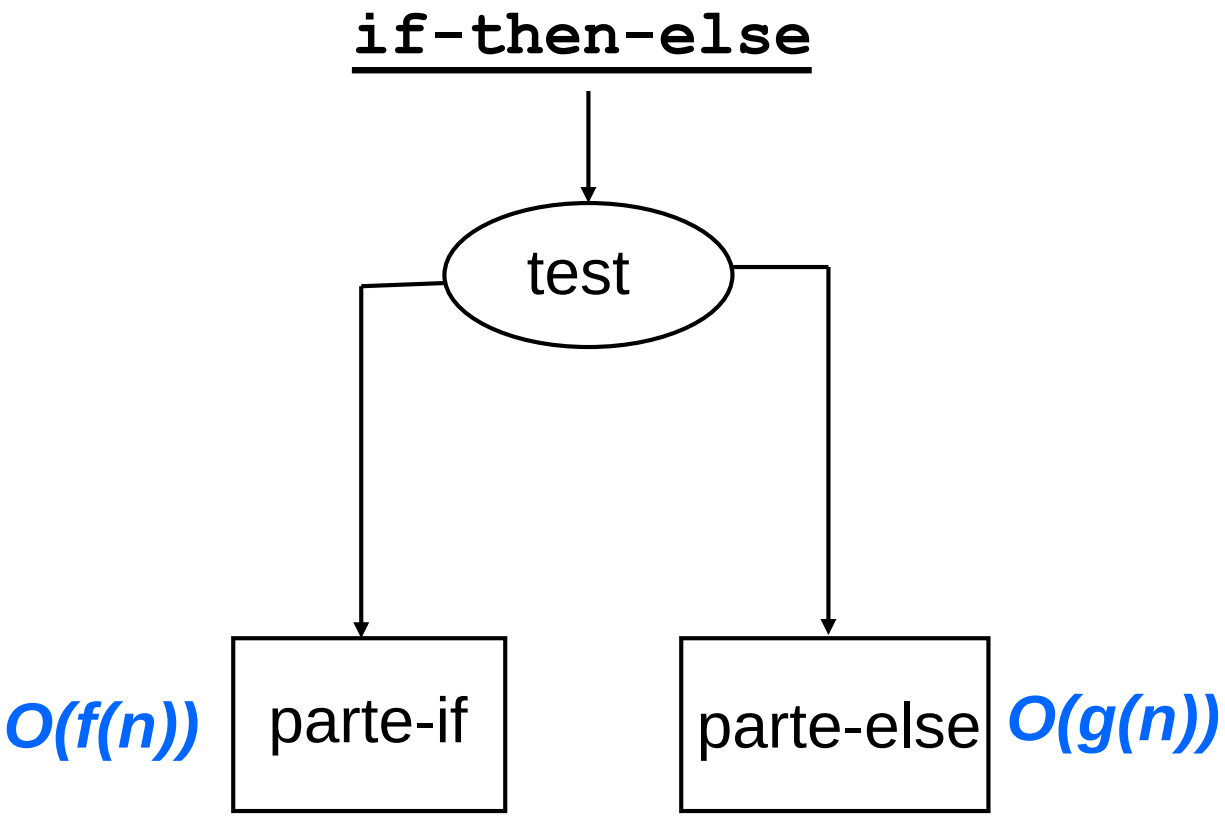
```
for i = 1 to n

    for j = i to n

        k = i + j
```

$= O(n - i)$

$= O(n^2)$

$T(n) = O(n \times n) = O(n^2)$

Tempo di esecuzione: If-Then-Else



$$O(\max(f(n), g(n)))$$

If-Then-Else: esempio

```
if A[1][i] = 0 then
    for i = 1 to n
        for j = 1 to n
        a[i][j] = 0
else
    for i = 1 to n
        A[i][i] = 1
```

$= O(n)\Big\} = O(n^2)$

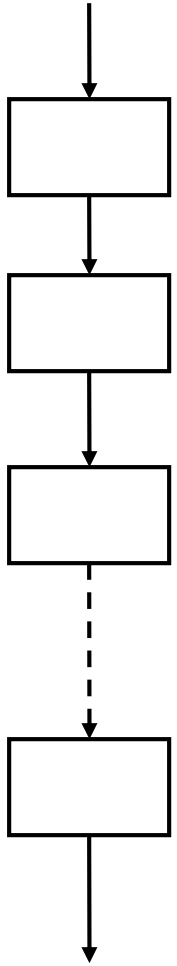
$= O(n)$

if: $T(n) = O(n^2)$

else : $T(n) = O(n)$

$T(n) = \max(O(n^2), O(n)) = O(n^2)$

Tempo di esecuzione: blocchi sequenziali



$O(f_1(n))$

$O(f_2(n))$

$O(f_3(n))$

$O(f_m(n))$

$$O(f_1(n) + f_2(n) + \ldots + f_m(n))$$

$$O\left(\max_{i \in \{1 \ldots m\}} \{f_i(n)\}\right)$$

Blocchi sequenziali: esempio

```
for i = 1 to n
  A[1] = 0
```
$= O(n)$

```
for i = 1 to n
  for j = 1 to n
    A[i] = A[i] + A[i]
```
$= O(n) = O(n^2)$

$T(n) = O(\max(f(\text{ciclo-1}), f(\text{ciclo-2}))$

$= O(\max(n, n^2))$

$= O(n^2)$

Esempio: Insert Sort

$O(n^2) =$

```
InsertSort(array A[1…n])
   for j = 2 to n
      key = A[j]                          = O(1)
      i = j - 1                           = O(1)
      while i > 0 and A[i] > key
            A[i+1] = A[i]                 } = O(n)
            i = i - 1
      A[i+1] = key                        = O(1)
```

$T(n) = O(g(n) \times max(1, 1, n, 1))$

$= O(n \times n)$

$= O(n^2)$

Tecniche di sviluppo di algoritmi

- **Agli esempi visti fino ad ora seguono l'*approccio incrementale*: la soluzione viene costruita passo dopo passo.**
- ***Insert sort* avendo ordinato una sottoparte dell'array, inserisce al posto giusto un altro elemento ottenendo un sotto-array ordinato più grande.**
- **Esistono altre tecniche di sviluppo di algoritmi con filosofie differenti:**
  - **Divide-et-Impera**

Divide-et-Impera

- **Il problema viene suddiviso in sottoproblemi analoghi, che vengono risolti separatamente. Le soluzioni dei sottoproblemi vengono infine fuse insieme per ottenere la soluzione dei problemi più complessi.**
- **Consiste di 3 passi:**
  - ***Divide* il problema in vari sottoproblemi, tutti *simili* (tra loro e) al *problema originario* ma più semplici.**
  - ***Impera* (conquista) i sottoproblemi risolvendoli ricorsivamente. Quando un sottoproblema diviene banale, risolverlo direttamente.**
  - ***Fondi* le soluzioni dei sottoproblemi per ottenere la soluzione del (sotto)problema che li ha originati.**

Divide-et-Impera e ordinamento

- **Input: una sequenza di numeri.**
- **Output: una permutazione (riordinamento) tale che tra ogni 2 elementi adiacenti nella sequenza valga "qualche" relazione di ordinamento (ad es. ≤).**
- ***Merge Sort* (divide-et-impera)**
  - ***Divide:*** **scompone la sequenza di *n* elementi in 2 sottosequenze di *n*/2 elementi ciascuna.**
  - ***Impera:*** **conquista i sottoproblemi ordinando ricorsivamente le sottosequenze con *Merge Sort* stesso. Quando una sottosequenza è unitaria, il sottoproblema è banale.**
  - ***Fondi:*** **compone insieme le soluzioni dei sottoproblemi per ottenere la sequenza ordinata del (sotto-)problema.**

Merge Sort

Algoritmo :

- ***A*[1..*n*]**: sequenza dei numeri in input
- ***p,r***: indici degli estremi della sottosequenza da ordinare

```
Merge_Sort(array A, int p,r)
1   if p < r then
2         q = ⌊(p+r)/2⌋
3         Merge_Sort(A,p,q)
4         Merge_Sort(A,q+1,r)
5         Merge(A,p,q,r)
```

Divide

Impera

Combina

***Esercizio*: definire la procedure Merge**

Merge Sort: analisi

```
Merge_Sort(array A, int p,r)
1   if p < r then
2           q = ⌊(p+r)/2⌋
3           Merge_Sort(A,p,q)
4           Merge_Sort(A,q+1,r)
5           Merge(A,p,q,r)
```

$T(n) = \Theta(1)$ se $n=1$

$T(n) = 2\,T(n/2) + T_{merge}(n) + \Theta(1)$

$T_{merge}(n) = \Theta(n)$

Equazione di Ricorrenza

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

Merge Sort: analisi

```
Merge_Sort(array A, int p,r)
1  if p < r then
2          q = ⌊(p+r)/2⌋
3           Merge_Sort(A,p,q)
4           Merge_Sort(A,q+1,r)
5           Merge(A,p,q,r)
```

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

***Soluzione*:** $T(n) = \Theta(n \log n)$

Divide-et-Impera: Equazioni di ricorrenza

- ***Divide*: *D(n)* tempo per dividere il problema**
- ***Impera*: se si divide il problema in *a* sottoproblemi, ciascuno di dimensione *n/b*, il tempo per conquistare i sottoproblemi sarà *aT(n/b)*.**

  Quando un sottoproblema diviene banale (*l'input è minore o uguale ad una costante c*), in tempo è *Θ(1)*.
- ***Fondi*: *C(n)* tempo per comporre le soluzioni dei sottoproblemi nella soluzione più complessa.**

$$T(n) = \begin{cases} \Theta(1) & se\ n \le c \\ aT(n/b) + D(n) + C(n) & se\ n > c \end{cases}$$

Gli argomenti trattati

- **Analisi della bontà di un algoritmo**
  - **Correttezza, utilizzo delle risorse, semplicità**
- **Modello computazionali: modello RAM**
- **Tempo di esecuzione degli algoritmi**
- **Notazione asintotica: *O*-grande, Ω-grande, Θ**
- **Analisi del Caso Migliore, Caso Peggiore e del Caso Migliore**