Algoritmi e Strutture di Dati

Complessità degli algoritmi

m.patrignani

Nota di copyright

- queste slides sono protette dalle leggi sul copyright
- il titolo ed il copyright relativi alle slides (inclusi, ma non limitatamente, immagini, foto, animazioni, video, audio, musica e testo) sono di proprietà degli autori indicati sulla prima pagina
- le slides possono essere riprodotte ed utilizzate liberamente, non a fini di lucro, da università e scuole pubbliche e da istituti pubblici di ricerca
- ogni altro uso o riproduzione è vietata, se non esplicitamente autorizzata per iscritto, a priori, da parte degli autori
- gli autori non si assumono nessuna responsabilità per il contenuto delle slides, che sono comunque soggette a cambiamento
- questa nota di copyright non deve essere mai rimossa e deve essere riportata anche in casi di uso parziale

Algoritmi e programmi

- un *algoritmo* coincide per noi con la sua descrizione in pseudocodice
  - sappiamo che ciò è equivalente a definire una Random Access Machine
- lo pseudocodice può essere facilmente tradotto in un linguaggio di programmazione arbitrario per ottenere un *programma*
- l'operazione di traduzione di un algoritmo in un programma viene detta *implementazione*

Esecuzione dei programmi

- il programma può essere eseguito
  - su una piattaforma qualsiasi
  - con dati di input qualsiasi
- la sua esecuzione ha un costo (economico) che può essere espresso tramite le risorse di calcolo utilizzate
  - tempo
  - memoria
  - traffico generato su rete
  - trasferimento dati da/su disco
  - ....
- nella maggior parte dei casi la risorsa più critica è il tempo di calcolo

Tempo di calcolo e dimensione dell'input

- si riscontra che il tempo di calcolo cresce al crescere della dimensione *n* dell'input
  - è legittimo misurarlo come una funzione di *n*
- la nozione di dimensione dell'input dipende dal problema
  - ordinamento:
    - numero di elementi da ordinare
  - operazioni su liste:
    - lunghezza della lista
  - operazioni su matrici:
    - dimensione massima delle matrici coinvolte
  - valutazione di un polinomio in un punto:
    - grado del polinomio

Fattori che influenzano il tempo di calcolo

- dimensione dell'input
  - maggiore è la quantità di dati in input maggiore è il tempo necessario per processarli
- algoritmo
  - può essere più o meno efficiente
- hardware
  - un supercalcolatore è più veloce di un personal computer
- linguaggio
  - un'implementazione diretta in linguaggio macchina è più veloce di un'implementazione in un linguaggio ad alto livello
  - un programma compilato è più veloce di un programma interpretato
- compilatore
  - alcuni compilatori sono progettati per generare codice efficiente
- programmatore
  - a parità di algoritmo e di linguaggio, programmatori esperti scelgono costrutti più veloci

Tempo di calcolo e algoritmi

esperimento: confronto tra due algoritmi di ordinamento

- stesso input: 1.000.000 numeri interi

algoritmo	insertion sort	merge sort
hardware	supercalcolatore	personal computer
linguaggio	linguaggio macchina	linguaggio ad alto livello
compilatore	–	non efficiente
programmatore	esperto	medio

tempo	5,56 ore	16,67 minuti

conclusione:

- il tempo di calcolo di un programma su uno specifico input è influenzato dall'algoritmo che implementa più che dagli altri fattori

Progetto di algoritmi efficienti

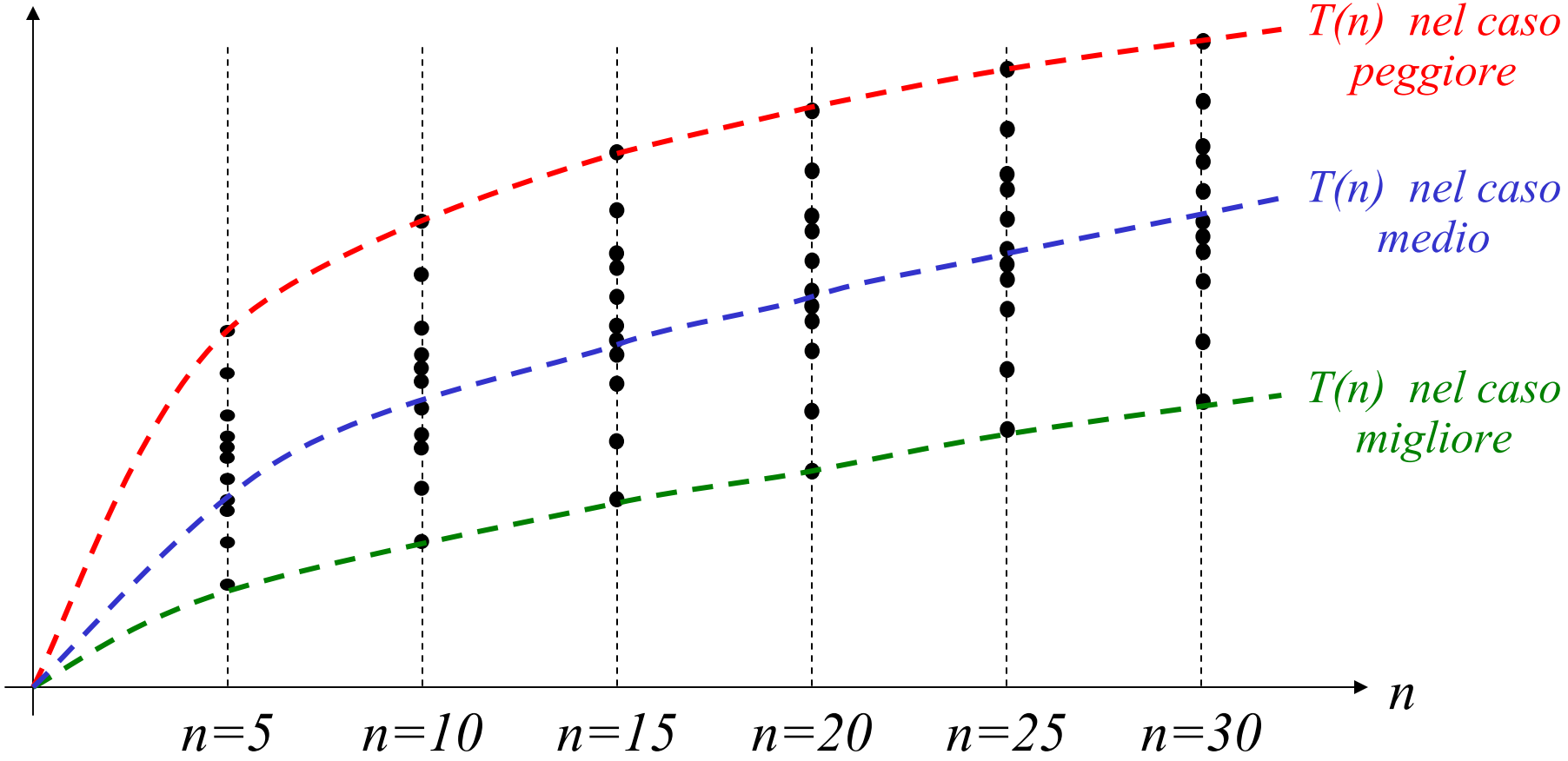
- motivazione
  - ha un impatto economico diretto per gli utilizzatori dei programmi
    - il tempo di calcolo si traduce in un investimento economico
  - è fondamentale per ottenere implementazioni di uso pratico
    - l'usabilità di un programma può essere compromessa da un algoritmo inefficiente
- problema
  - nel momento in cui progettiamo un algoritmo non possiamo misurare direttamente l'efficienza delle sue future implementazioni
- soluzione
  - previsione del tempo di calcolo delle implementazioni di un algoritmo (analisi)

Analisi degli algoritmi

- obiettivo
  - prevedere il tempo di calcolo richiesto dall'esecuzione di un programma che implementa il nostro algoritmo
    - in funzione della dimensione dell'input
      - per piccoli input il tempo di calcolo sarà comunque basso
      - qual è il tempo di calcolo per input di grandi dimensioni?
- strumenti
  - ipotesi sul tempo di esecuzione di ogni istruzione
  - analisi asintotica delle funzioni
    - quale funzione consideriamo?

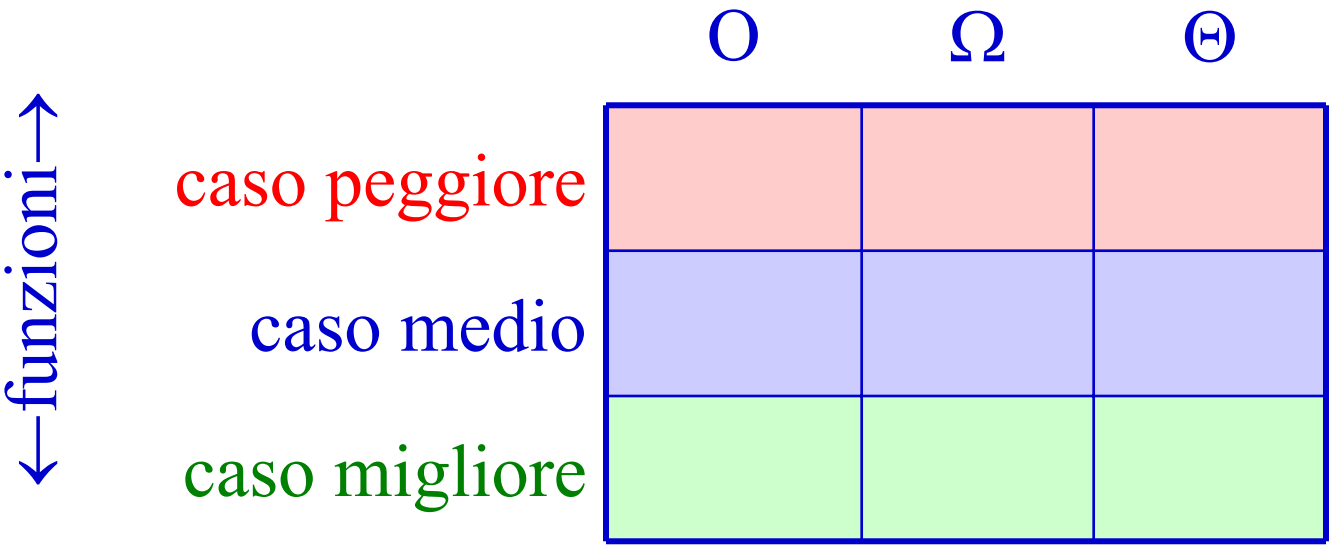
Il tempo di calcolo non è una funzione

- In generale il tempo di calcolo per un input di dimensione *n* **non è** una funzione



Tempo di calcolo e analisi asintotica

- vogliamo studiare il tempo di calcolo con gli strumenti dell'analisi asintotica
  - ma l'analisi asintotica si applica solo alle funzioni
  - dobbiamo trasformare il tempo di calcolo in una funzione
  - per questo consideriamo il caso peggiore/medio/migliore



Uso del caso peggiore

- in generale è di maggiore interesse il *caso peggiore* rispetto al *caso migliore* o al *caso medio*
  - si preferisce un errore per eccesso ad un errore per difetto
  - il caso migliore non dà nessuna garanzia sul tempo di calcolo con un input generico
    - è di interesse solamente teorico
  - spesso il tempo di calcolo del caso medio è più vicino al caso peggiore che al caso migliore
  - conoscere il costo del caso medio può essere utile solo qualora si debba ripetere un'operazione un numero elevato di volte

Stima del tempo di calcolo

- denotiamo *T(n)* il tempo di calcolo di una implementazione dell'algoritmo nel caso peggiore su un input di dimensione *n*
- vogliamo stimare *T(n)* a partire dallo pseudocodice
- quanto costa ogni operazione elementare?
- ipotesi semplificativa:
  - per eseguire una linea (o istruzione) di pseudocodice è richiesto tempo costante
  - denotiamo con c_i il tempo necessario per eseguire la riga *i*

Strategie per la stima del tempo di calcolo

- strategia più onerosa
  - calcoliamo esplicitamente T(n) a partire dallo pseudocodice
    - T(n) dipende, oltre dalla dimensione dell'input n, anche dal costo di esecuzione associato alle singole righe dello pseudocodice c_1, c_2, c_3, c_4, …
  - studiamo il comportamento asintotico di T(n)
- strategia più veloce
  - calcoliamo il costo asintotico di ogni porzione dello pseudocodice
  - otteniamo il costo asintotico dell'intero algoritmo componendo i costi calcolati
  - otteniamo il comportamente asintotico di T(n) senza mai calcolare esplicitamente T(n)

Esempio di algoritmo

- algoritmo DUE_OCCORRENZE
  - accetta in input un array di interi
  - ritorna TRUE se esiste un valore che occorre almeno due volte
  - ritorna FALSE se tutti gli elementi sono diversi

```
DUE_OCCORRENZE(A)
1.  output = false
2.  for i = 0 to A.length-2
3.      for j = i+1 to A.length-1
4.          if (A[i] == A[j])
5.              output = true
6.  return output
```


Esempio di calcolo di T(n)

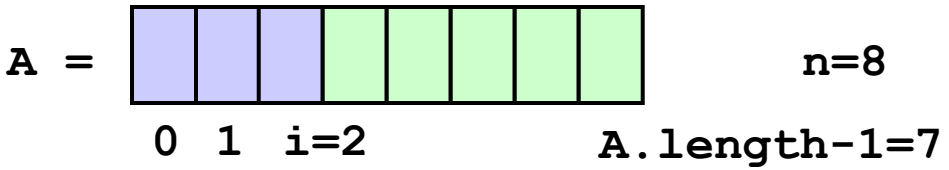
DUE_OCCORRENZE(A)	costo	n° di volte
1. output = **false**	c_1	1
2. **for** i = 0 **to** A.length-2	c_2	
3. **for** j = i+1 **to** A.length-1	c_3	
4. **if** (A[i] == A[j])	c_4	
5. output = **true**	c_5	
6. **return** output	c_6	1

supponiamo che ogni riga i abbia un costo di esecuzione c_i

contiamo quante volte ogni riga viene eseguita

Esempio di calcolo di T(n)

DUE_OCCORRENZE(A)	costo	n° di volte
1. output = **false**	c_1	1
2. **for** i = 0 **to** A.length-2	c_2	n
3. **for** j = i+1 **to** A.length-1	c_3	
4. **if** (A[i] == A[j])	c_4	
5. output = **true**	c_5	
6. **return** output	c_6	1



- il ciclo esterno viene eseguito eseguito n-1 volte
  - il test, dunque, viene eseguito n volte
    - l'ultimo test determina l'uscita dal ciclo for
- il ciclo interno viene eseguito n-i-1 volte

Esempio di calcolo di T(n)

DUE_OCCORRENZE(A)	costo	n° di volte
1. output = **false**	c_1	1
2. **for** i = 0 **to** A.length-2	c_2	n
3. **for** j = i+1 **to** A.length-1	c_3	
4. **if** (A[i] == A[j])	c_4	
5. output = **true**	c_5	
6. **return** output	c_6	1

- per ogni i, il ciclo interno viene eseguito n-i-1 volte, cioè: $\Sigma_{i=0..n-2}(n\text{-}i\text{-}1) = \Sigma_{i=0..n-2}(n) - \Sigma_{i=0..n-2}(i) - \Sigma_{i=0..n-2}(1)$
- dove

$\Sigma_{i=0..n-2}(n) = (n\text{-}1)n$

$\Sigma_{i=0..n-2}(i) = (n\text{-}1)(n\text{-}2)/2$ ←

$\Sigma_{i=0..n-2}(1) = (n\text{-}1)$

formula di Gauss

$$\Sigma_{i=1..n}(i) = \frac{(n+1)n}{2}$$

Esempio di calcolo di T(n)

DUE_OCCORRENZE(A)	costo	n° di volte
1. output = **false**	c_1	1
2. **for** i = 0 **to** A.length-2	c_2	n
3. **for** j = i+1 **to** A.length-1	c_3	$0.5n^2+1.5n+1$
4. **if** (A[i] == A[j])	c_4	$0.5n^2+1.5n$
5. output = **true**	c_5	$0.5n^2+1.5n$
6. **return** output	c_6	1

- il ciclo interno viene eseguito $(n-1)n - (n-1)(n-2)/2 - (n-1) = n^2-n - (n^2-3n+2)/2 - n+1 = n^2-n -0.5n^2+1.5n-1+n+1 = 0.5n^2+1.5n$ volte
- il test alla riga 2 viene eseguito $0.5n^2+1.5n+1$ volte

Esempio di calcolo di T(n)

DUE_OCCORRENZE(A)	costo	n° di volte
1. output = **false**	c_1	1
2. **for** i = 0 **to** A.length-2	c_2	n
3. **for** j = i+1 **to** A.length-1	c_3	$0.5n^2+1.5n+1$
4. **if** (A[i] == A[j])	c_4	$0.5n^2+1.5n$
5. output = **true**	c_5	$0.5n^2+1.5n$
6. **return** output	c_6	1

$$T(n) = c_1+c_2(n)+c_3(0.5n^2+1.5n+1)+(c_4+c_5)(0.5n^2+1.5n)+c_6$$
$$= n^2\,(0.5c_3+0.5c_4+0.5c_5) + n\,(c_2+1.5c_3+1.5c_4+1.5c_5) + (c_1+c_3+c_6)$$

Dunque: $T(n) \in O(n^2)$, $T(n) \in \Omega(n^2) \Rightarrow T(n) \in \Theta(n^2)$

Algoritmi e complessità O($f(n)$)

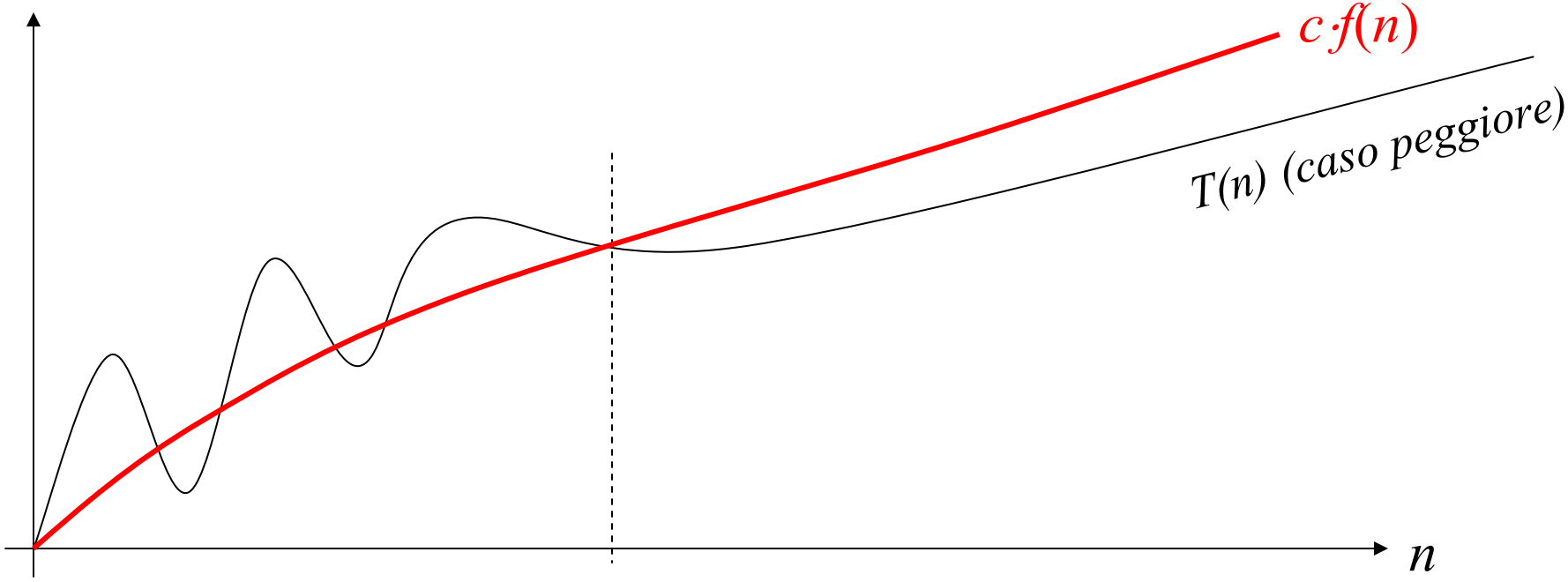
- sia T(n) il tempo di esecuzione di un algoritmo A su un'istanza di dimensione n nel caso peggiore

> l'algoritmo A ha *complessità temporale O(f(n))* se
> $T(n) \in O(f(n))$

- diciamo anche che
  - il tempo di esecuzione dell'algoritmo A è *al più f(n)*
  - $f(n)$ è un *limite superiore*, o *upper-bound*, al tempo di esecuzione dell'algoritmo A
  - $f(n)$ è la quantità di tempo *sufficiente* (in ogni caso) all'esecuzione dell'algoritmo A

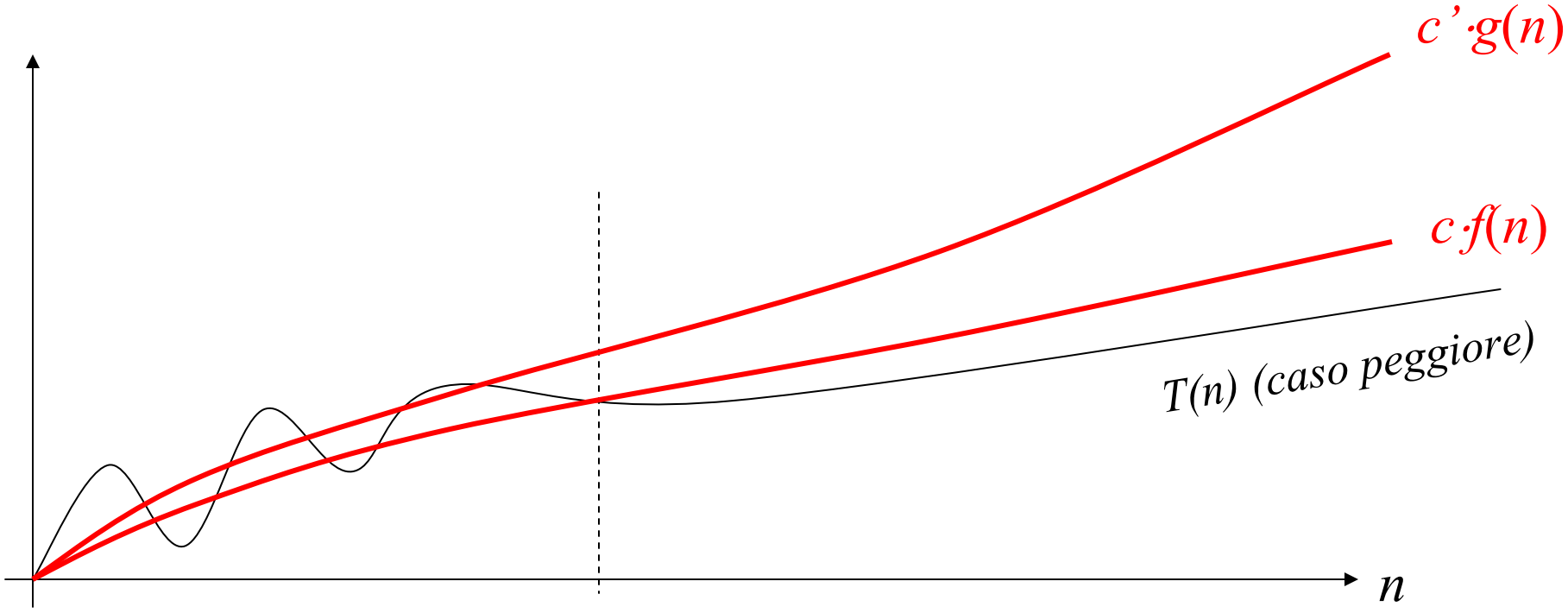
Algoritmi e complessità O($f(n)$)

l'algoritmo A ha *complessità temporale O(f(n))* se
$T(n) \in O(f(n))$



Significatività della funzione $f(n)$

- se un algoritmo ha complessità $f(n)$ allora ha anche complessità $g(n)$ per ogni $g(n)$ tale che $f(n) \in O(g(n))$
- la complessità espressa tramite la notazione O-grande diventa tanto più significativa quanto più $f(n)$ è stringente (piccolo)



Algoritmi e complessità $\Omega(f(n))$

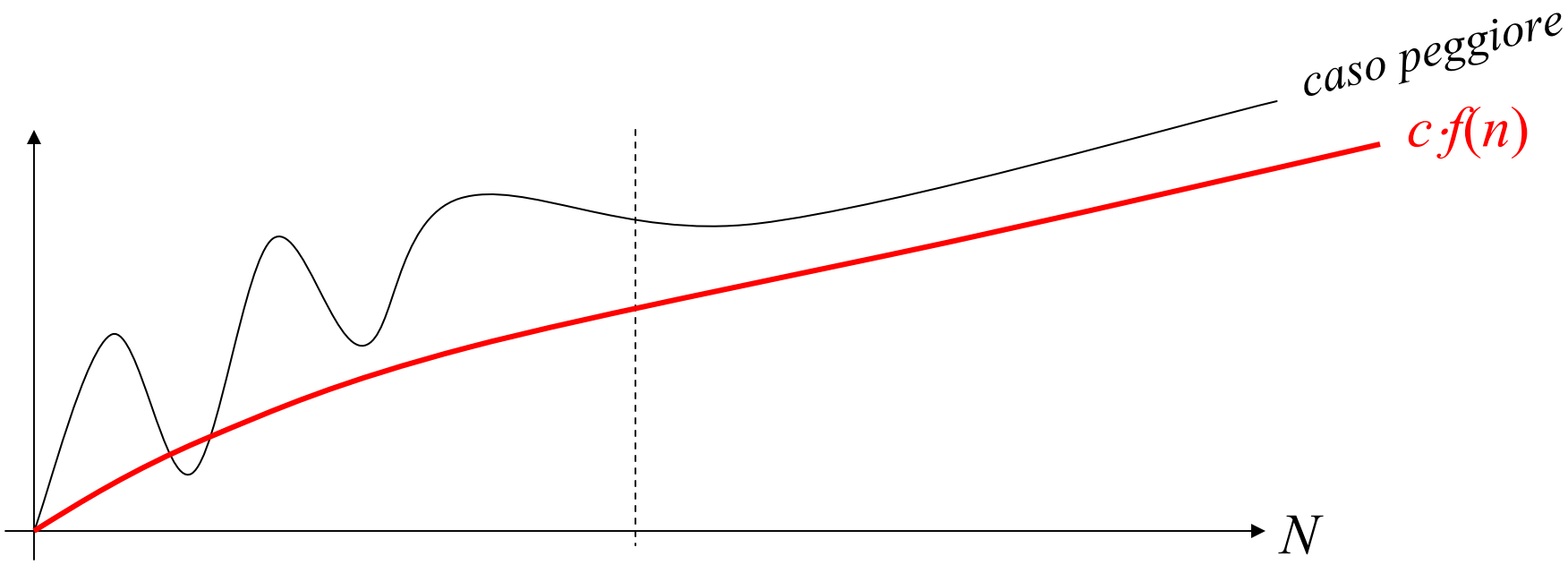
- sia $T(n)$ il tempo di esecuzione di un algoritmo A su un'istanza di dimensione n nel caso peggiore

> l'algoritmo A ha *complessità temporale* $\Omega(f(n))$ se $T(n) \in \Omega(f(n))$

- diciamo anche che
  - il tempo di esecuzione dell'algoritmo A è *almeno* $f(n)$
  - $f(n)$ è un *limite inferiore*, o *lower-bound*, al tempo di esecuzione dell'algoritmo A
  - $f(n)$ è la quantità di tempo *necessaria* (in almeno un caso) all'esecuzione dell'algoritmo A

Algoritmi e complessità $\Omega(f(n))$

l'algoritmo *A* ha *complessità temporale* $\Omega(f(n))$ se
$T(n) \in \Omega(f(n))$



Algoritmi e complessità $\Theta(f(n))$

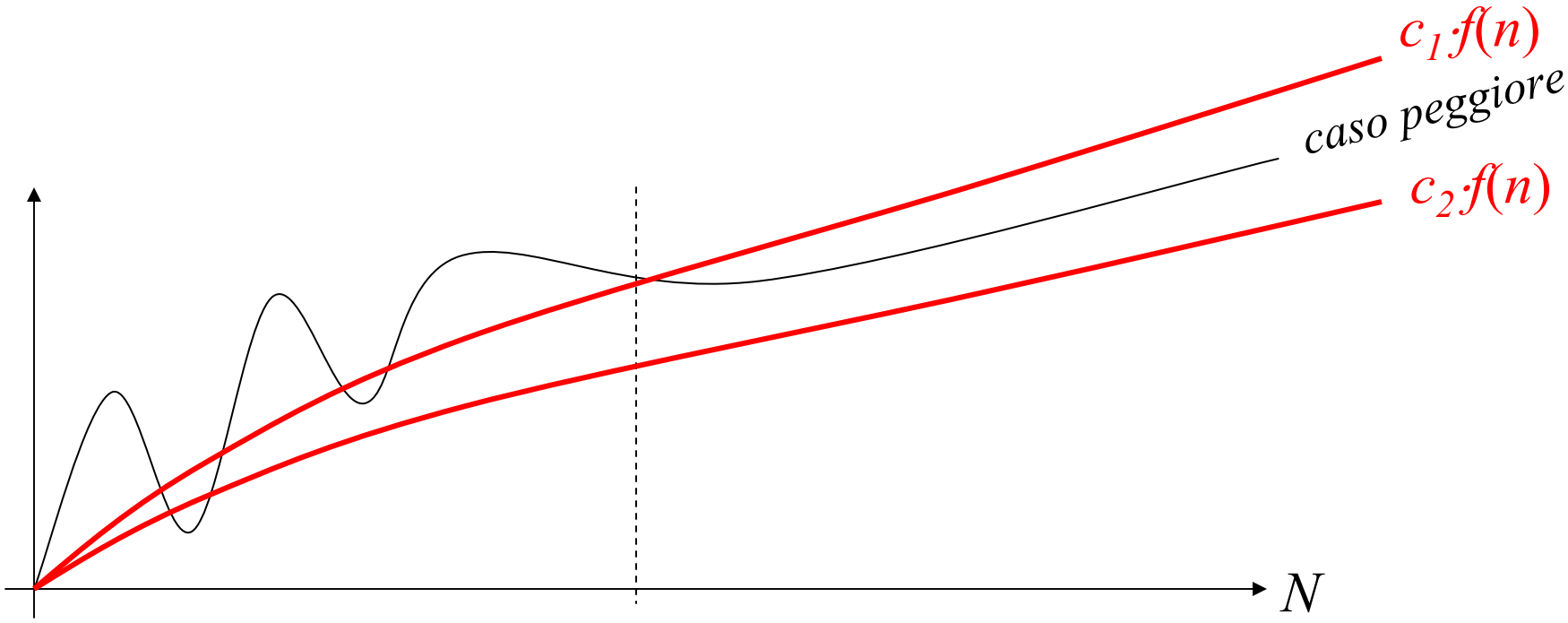
- sia $T(n)$ il tempo di esecuzione di un algoritmo A su un'istanza di dimensione n nel caso peggiore

> l'algoritmo A ha *complessità temporale* $\Theta(f(n))$ se ha complessità temporale $O(f(n))$ e $\Omega(f(n))$

- diciamo anche che
  - il tempo di esecuzione dell'algoritmo è $f(n)$
  - $f(n)$ è un *limite inferiore* e *superiore* (*lower-bound* e *upper-bound*), al tempo di esecuzione dell'algoritmo
  - $f(n)$ è la quantità di tempo *necessaria e sufficiente* all'esecuzione dell'algoritmo

Algoritmi e complessità $\Theta(f(n))$

l'algoritmo *A* ha *complessità temporale* $\Theta(f(n))$ se
$T(n) \in \Theta(f(n))$



Strategie di analisi più efficienti

DUE_OCCORRENZE(A)	n° di volte	ordine
1. output = **false**	1	$\Theta(1)$
2. **for** i = 0 **to** A.length-2	n	$\Theta(n)$
3. **for** j = i+1 **to** A.length-1	$0.5n^2+1.5n+1$	$\Theta(n^2)$
4. **if** (A[i] == A[j])	$0.5n^2+1.5n$	$\Theta(n^2)$
5. output = **true**	$0.5n^2+1.5n$	$\Theta(n^2)$
6. **return** output	1	$\Theta(1)$

- la regola della somma mi assicura che l'andamento asintotico di una somma di termini può essere ottenuto considerando l'andamento asintotico dei singoli termini
- una strategia più efficiente si basa, dunque, sul calcolo diretto del costo asintotico di ogni riga

Calcolo efficiente del costo asintotico

- istruzioni semplici
  - tempo di esecuzione costante: $\Theta(1)$
- sequenza (necessariamente finita) di istruzioni semplici
  - tempo di esecuzione costante: $\Theta(1)$
- sequenza di istruzioni generiche
  - somma dei tempi di esecuzione di ciascuna istruzione

Istruzioni condizionali

```
1.  if <condizione> then
2.      <parte-then>
3.   else
4.      <parte-else>
```

per calcolare T(n) occorrerebbe sapere se la condizione si verifica o meno

- troviamo un limite superiore O-grande al tempo di esecuzione T(n) come somma dei costi seguenti
  - costo O-grande della valutazione della condizione
  - costo O-grande maggiore tra `<parte-then>` e `<parte-else>`
- troviamo un limite inferiore Ω al tempo di esecuzione T(n) come somma dei costi seguenti
  - costo Ω della valutazione della condizione
  - costo Ω minore tra `<parte-then>` e `<parte-else>`

Istruzioni ripetitive

- il nostro pseudocodice ci offre tre istruzioni ripetitive
  - **`for`**, **`while`** e **`repeat`**
- per il limite superiore O-grande occorre determinare
  - un limite superiore $O(f(n))$ al numero di iterazioni del ciclo
  - un limite superiore $O(g(n))$ al tempo di esecuzione di ogni iterazione
    - si compone del costo dell'esecuzione del blocco di istruzioni più il costo di esecuzione del test
- il costo del ciclo sarà: $O(g(n) \cdot f(n))$
- analogamente sarà: $\Omega(g'(n) \cdot f'(n))$
  - dove le iterazioni sono $\Omega(g'(n))$ ed il costo di una iterazione è $\Omega(f'(n))$

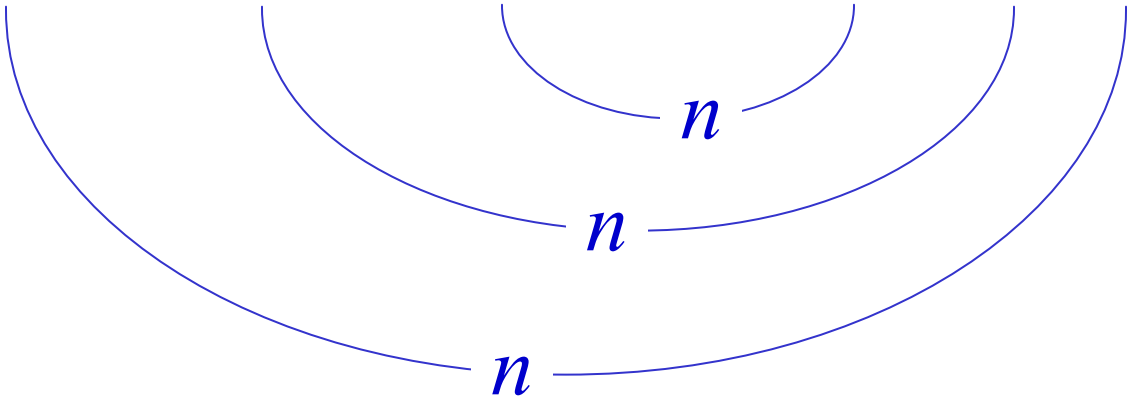
Istruzioni ripetitive annidate

```
2.  for i = 0 to n
3.      for j = i+1 to n
4.          <istruzione o blocco>
```

Un caso frequente è quando il secondo ciclo viene eseguito meno volte del primo

Quante volte viene eseguita l'istruzione?

$$(n\text{-}1) + (n\text{-}2) + (n\text{-}3) + \ldots + 3 + 2 + 1$$



$$n/2 \times n = n^2/2 = \Theta(n^2)$$

Istruzioni ripetitive: esempio

```
FACT(n)
1.  f = 1
2.  for k = 2 to n
3.      f = f * k
4.  return f
```

- numero di iterazioni del ciclo for: $\Theta(n)$
- costo di una singola iterazione: $\Theta(1)$
- costo complessivo del ciclo while: $\Theta(n \cdot 1) = \Theta(n)$
- costo complessivo della procedura: $\Theta(n)$

Attenzione al modello!

```
FACT(n)
1.  f = 1
2.  for k = 2 to n
3.      f = f * k
4.  return f
```

- stiamo lavorando nell'ipotesi in cui le variabili (che corrispondono ai registri della RAM) riescano sempre a contenere i numeri coinvolti
- se usassimo come misura dell'input il numero k di bit necessari per rappresentare n in binario avremmo $k = \lceil \log_2 n \rceil$ e costo complessivo = $\Theta(2^k)$

Chiamata a funzione o procedura

```
1.  P(…)
2.      …
3.      Q(…)
4.      …
```

supponiamo che un programma P invochi la procedura Q

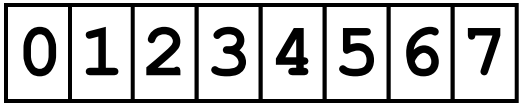
- sia $T_Q(n)$ il tempo di esecuzione della procedura Q
- il tempo di esecuzione dell'invocazione della procedura Q in P è $T_Q(m)$, dove m è la dimensione dell'input passato alla procedura Q
  - attenzione: occorre determinare la relazione tra m e la dimensione n dell'input di P

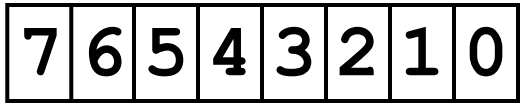
Esempio di chiamata a funzione

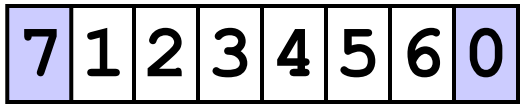
```
SUM_OF_FACT(n)
1.  sum = 0
2.  for m = 0 to n
3.      sum = sum + FACT(m)
4.  return sum
```

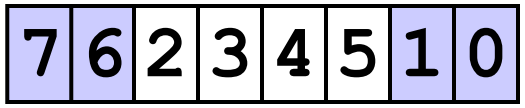
- il corpo del ciclo for ha complessità $\Theta(1) + \Theta(m) = \Theta(m)$
- il ciclo viene eseguito n volte
- il costo complessivo del ciclo è dunque: $\Theta(n) + \Theta(n\text{-}1) + \ldots + \Theta(2) + \Theta(1) = \Theta(n^2)$
- il costo totale è $\Theta(n^2)$

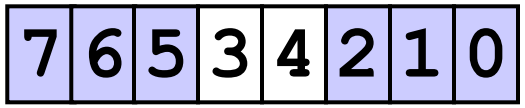
Esempio di analisi della complessità

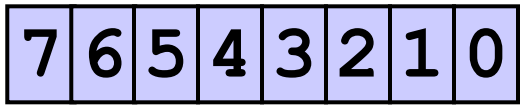
- algoritmo per invertire un array
  - da 

a 

- strategia
  - scambio A[0] con A[A.length-1] 

  - scambio A[1] con A[A.length-2] 

  - scambio A[2] con A[A.length-3] 

  - scambio A[3] con A[A.length-4] 

  - ...

Esempio di analisi della complessità

```
INVERTI_ARRAY(A)
1.  for i = 0 to ⌊A.length/2⌋ do
2.      SCAMBIA(A,i,A.length-1-i)
```

```
SCAMBIA(A,j,k)
1.  memo = A[j]
2.  A[j] = A[k]
3.  A[k]= memo
```

- la funzione **SCAMBIA**(A,j,k) ha complessità $\Theta(1)$ in quanto è composta da una successione di istruzioni elementari
- la funzione **INVERTI_ARRAY**(A) ha complessità $\Theta(n)$ in quanto esegue per $\Theta(n)$ volte il blocco delle istruzioni che consiste nell'esecuzione di una procedura $\Theta(1)$