

# algoritmi e strutture di dati

## complessità degli algoritmi

*m.patrignani*

040-complessita-algoritmi-05 copyright ©2014 patrignani@dia.uniroma3.it

## 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

040-complessita-algoritmi-05 copyright ©2014 patrignani@dia.uniroma3.it

## 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*

040-complessita-algoritmi-05 copyright ©2014 patrignani@dia.uniroma3.it

## esecuzione dei programmi

- il programma può essere eseguito
  - su una piattaforma opportuna
  - con dati di input opportuni
- 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

040-complessita-algoritmi-05 copyright ©2014 patrignani@dia.uniroma3.it

### 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

040-complessita-algoritmi-05 copyright ©2014 patrignani@dia.uniroma3.it

### 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

040-complessita-algoritmi-05 copyright ©2014 patrignani@dia.uniroma3.it

## 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:

– a parità di input, il tempo di calcolo di un programma è influenzato dall'algoritmo che implementa più che dagli altri fattori

040-complessita-algoritmi-05 copyright ©2014 patrignani@dia.uniroma3.it

## 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)

040-complessita-algoritmi-05 copyright ©2014 patrignani@dia.uniroma3.it

## analisi degli algoritmi

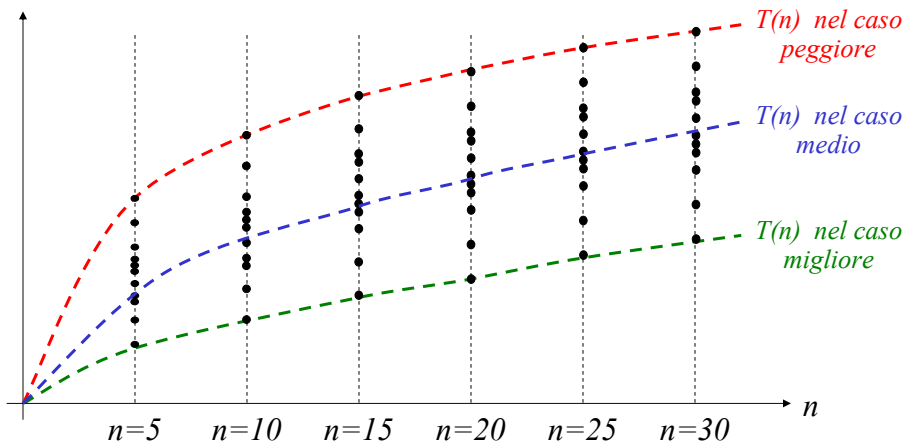
- obiettivo
  - prevedere il tempo di calcolo richiesto dall'esecuzione di un programma che implementa il nostro algoritmo
    - in funzione della dimensione dell'input
      - per piccoli input il tempo di calcolo sarà comunque basso
      - qual è il tempo di calcolo per input di grandi dimensioni?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

040-complessita-algoritmi-05

copyright ©2014 patrignani@dia.uniroma3.it

## il tempo di calcolo non è una funzione

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



040-complessita-algoritmi-05

copyright ©2014 patrignani@dia.uniroma3.it

## 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

		$O$	$\Omega$	$\Theta$
funzioni ↑ ↓	caso peggiore			
	caso medio			
	caso migliore			

040-complessita-algoritmi-05

copyright ©2014 patrignani@dia.uniroma3.it

## uso del caso peggiore

- per noi è di maggiore interesse il *caso peggiore* rispetto al *caso migliore* o al *caso medio*
  - preferiamo 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

040-complessita-algoritmi-05

copyright ©2014 patrignani@dia.uniroma3.it

### 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  $i$

040-complessita-algoritmi-05 copyright ©2014 patrignani@dia.uniroma3.it

### 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  $c_1, c_2, c_3, c_4, \dots$
  - studiamo il comportamento asintotico di  $T(n)$
- strategia più efficiente
  - calcoliamo il costo asintotico di ogni porzione dello pseudocodice
  - otteniamo il costo asintotico dell'intero algoritmo componendo i costi calcolati
  - otteniamo il comportamento asintotico di  $T(n)$  senza mai calcolare esplicitamente  $T(n)$

040-complessita-algoritmi-05 copyright ©2014 patrignani@dia.uniroma3.it

## esempio di algoritmo

- algoritmo per invertire un array

– da 

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

 a 

7	6	5	4	3	2	1	0
---	---	---	---	---	---	---	---

- strategia

– memorizzo A[0]

memo 

0
---

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

– copio A[A.length-1] in A[0]

memo 

0
---

7	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

– traslo tutte le caselle A[1..A.length-2] in avanti

memo 

0
---

7		1	2	3	4	5	6
---	--	---	---	---	---	---	---

– copio memo in A[1]

memo 

0
---

7	0	1	2	3	4	5	6
---	---	---	---	---	---	---	---

– ripeto per A[1]

7	0	1	2	3	4	5	6
---	---	---	---	---	---	---	---

7	6	0	1	2	3	4	5
---	---	---	---	---	---	---	---

– ripeto per A[2]

7	6	0	1	2	3	4	5
---	---	---	---	---	---	---	---

7	6	5	0	1	2	3	4
---	---	---	---	---	---	---	---

– ...

040-complessita-algoritmi-05

copyright ©2014 patrignani@dia.uniroma3.it

## esempio di calcolo di $T(n)$

INVERTI-ARRAY (A)	costo	n° di volte
1. <b>for</b> i = 0 <b>to</b> A.length-2	$c_1$	n
2.     memo = A[i]	$c_2$	n-1
3.     A[i] = A[A.length-1]	$c_3$	n-1
4. <b>for</b> j = A.length-1 <b>down to</b> i+2	$c_4$	
5.         ▷ traslo in avanti A[i+1..]	0	
6.         A[j] = A[j-1]	$c_6$	
7.     A[i+1] = memo	$c_7$	n-1

- 1     il test viene eseguito  $n$  volte

- n-1 volte entro nel ciclo for
- l'ultimo test determina l'uscita dal ciclo

2,3,7 linee eseguite per ogni iterazione del ciclo for, cioè  $n-1$  volte

040-complessita-algoritmi-05

copyright ©2014 patrignani@dia.uniroma3.it



## esempio di calcolo di $T(n)$

INVERTI-ARRAY (A)	costo	n° di volte
1. <b>for</b> i = 0 <b>to</b> A.length-2	$c_1$	n
2.     memo = A[i]	$c_2$	n-1
3.     A[i] = A[A.length-1]	$c_3$	n-1
4. <b>for</b> j = A.length-1 <b>down to</b> i+2	$c_4$	$(n-1)(n+2)/2$
5.         ▷ traslo in avanti A[i+1..]	0	$(n-1)(n+2)/2-1$
6.         A[j] = A[j-1]	$c_6$	$(n-1)(n+2)/2-1$
7.         A[i+1] = memo	$c_7$	n-1

4 il test viene eseguito, per ogni  $i = 0, \dots, n-2$  (cioè  $n-1$  volte) un numero di volte pari a  $n-(i+2)+2=n-i$ , quindi  $\sum_{i=0..n-2}(n-i) = \sum_{i=0..n-2}(n) - \sum_{i=0..n-2}(i) = (n-1)n - (n-2)(n-1)/2 = (n-1)(n+2)/2$

• si è utilizzata la formula di Gauss  $\sum_{i=1..n}(i) = (n+1)n/2$

5,6 come sopra -1

## esempio di calcolo di $T(n)$

INVERTI-ARRAY (A)	costo	n° di volte
1. <b>for</b> i = 0 <b>to</b> A.length-2	$c_1$	n
2.     memo = A[i]	$c_2$	n-1
3.     A[i] = A[A.length-1]	$c_3$	n-1
4. <b>for</b> j = A.length-1 <b>down to</b> i+2	$c_4$	$(n-1)(n+2)/2$
5.         ▷ traslo in avanti A[i+1..]	0	$(n-1)(n+2)/2-1$
6.         A[j] = A[j-1]	$c_6$	$(n-1)(n+2)/2-1$
7.         A[i+1] = memo	$c_7$	n-1

$$\begin{aligned}
 T(n) &= c_1n + c_2(n-1) + c_3(n-1) + c_4(n-1)n/2 + c_6((n-1)n/2-1) + c_7(n-1) \\
 &= n^2(c_4/2 + c_6/2) + \\
 &\quad n(c_1 + c_2 + c_3 - c_4/2 - c_6/2 + c_7) - \\
 &\quad (c_2 + c_3 + c_6 + c_7)
 \end{aligned}$$

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

## algoritmi e complessità $O(f(n))$

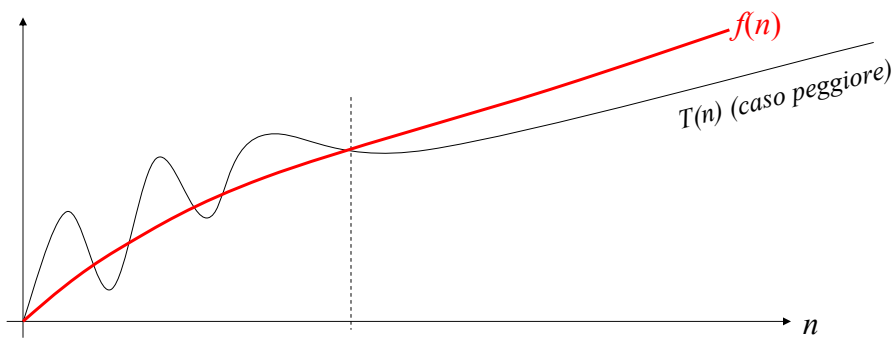
- sia  $T(n)$  il tempo di esecuzione di un algoritmo  $A$  su un'istanza di dimensione  $n$  nel caso peggiore

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

- 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  $A$

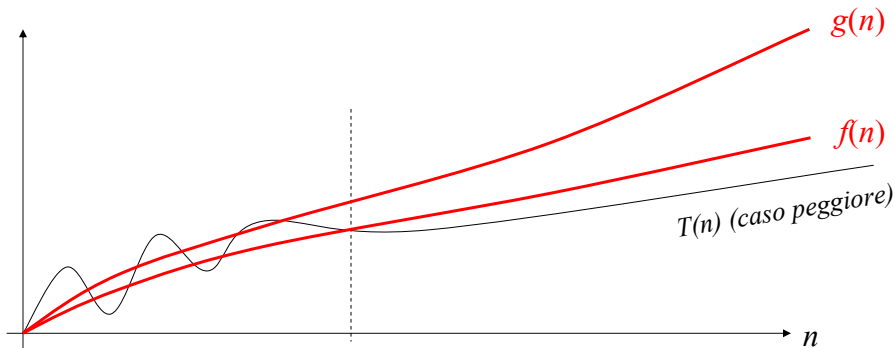
## algoritmi e complessità $O(f(n))$

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



## 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)



040-complessita-algoritmi-05 copyright ©2014 patrignani@dia.uniroma3.it

## 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  

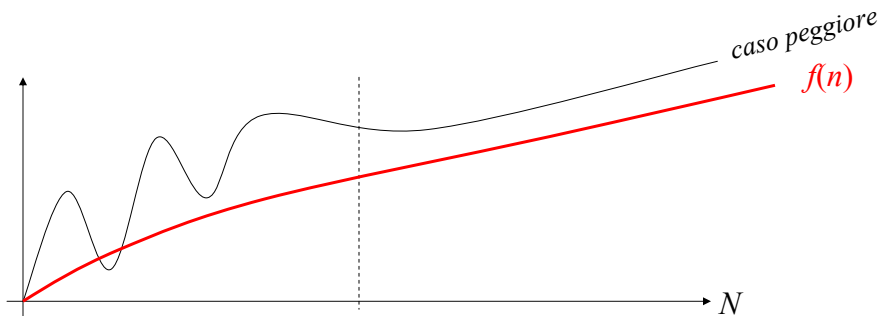
$$T(n) = \Omega(f(n))$$

- 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  $A$

040-complessita-algoritmi-05 copyright ©2014 patrignani@dia.uniroma3.it

## algoritmi e complessità $\Omega(f(n))$

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



040-complessita-algoritmi-05 copyright ©2014 patrignani@dia.uniroma3.it

## 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

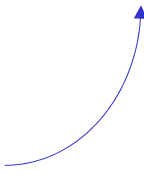
l'algoritmo  $A$  ha *complessità temporale*  $\Theta(f(n))$  se ha  
 complessità temporale  $O(f(n))$  e  $\Omega(f(n))$

- 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

040-complessita-algoritmi-05 copyright ©2014 patrignani@dia.uniroma3.it

## strategie di analisi più efficienti

INVERTI-ARRAY (A)	n° volte	ordine
1. <b>for</b> i = 0 <b>to</b> A.length-2	n	$\Theta(n)$
2.     memo = A[i]	n-1	$\Theta(n)$
3.     A[i] = A[A.length-1]	n-1	$\Theta(n)$
4. <b>for</b> j = A.length-1 <b>down to</b> i+2	$(n-1)(n+2)/2$	$\Theta(n^2)$
5.         ▷ traslo in avanti A[i+1..]	$(n-1)(n+2)/2-1$	$\Theta(n^2)$
6.         A[j] = A[j-1]	$(n-1)(n+2)/2-1$	$\Theta(n^2)$
7.     A[i+1] = memo	n-1	$\Theta(n)$

- una prima strategia più efficiente si ottiene trascurando le costanti  $c_1, c_2, c_3, \dots$  nel calcolo del costo di ogni riga
  - una strategia ancora più efficiente si basa sul calcolo diretto del costo asintotico di ogni riga
- 

040-complessita-algoritmi-05    copyright ©2014 patrignani@dia.uniroma3.it

## calcolo efficiente del costo asintotico

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

040-complessita-algoritmi-05    copyright ©2014 patrignani@dia.uniroma3.it

## 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  $\langle$ parte-then $\rangle$  e  $\langle$ parte-else $\rangle$
- troviamo un limite inferiore  $\Omega$  al tempo di esecuzione  $T(n)$  come somma dei costi seguenti
  - costo  $\Omega$  della valutazione della condizione
  - costo  $\Omega$  minore tra  $\langle$ parte-then $\rangle$  e  $\langle$ parte-else $\rangle$

040-complessita-algoritmi-05 copyright ©2014 patrignani@dia.uniroma3.it

## 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  $\Omega(f'(n))$

040-complessita-algoritmi-05 copyright ©2014 patrignani@dia.uniroma3.it

## istruzioni ripetitive: esempio

**FACT (n)**

```

1. f = 1
2. k = n
3. while k > 0
4.     do f = f * k
5.     k = k - 1
6. return f

```

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

040-complessita-algoritmi-05    copyright ©2014 patrignani@dia.uniroma3.it

## attenzione al modello

**FACT (n)**

```

1. f = 1
2. k = n
3. while k > 0
4.     do f = f * k
5.     k = k - 1
6. return f

```

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

040-complessita-algoritmi-05    copyright ©2014 patrignani@dia.uniroma3.it

## 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.	m = n
3.	<b>while</b> m > 0
4.	<b>do</b> sum = sum + <b>FACT</b> (m)
5.	m = m - 1
6.	<b>return</b> sum

- il corpo del ciclo while ha complessità  $\Theta(1) + \Theta(m) = \Theta(m)$
- il ciclo viene eseguito  $n$  volte, per i valori di  $m = n, n-1, \dots, 1$
- il costo complessivo del ciclo è dunque:  
 $\Theta(n) + \Theta(n-1) + \dots + \Theta(2) + \Theta(1) = \Theta(n^2)$
- il costo totale è  $\Theta(n^2)$



## esempio di analisi della complessità

- secondo algoritmo per invertire un array

– da 

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

 a 

7	6	5	4	3	2	1	0
---	---	---	---	---	---	---	---

- strategia

– scambio  $A[0]$  con  $A[A.length-1]$ 

7	1	2	3	4	5	6	0
---	---	---	---	---	---	---	---

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

7	6	2	3	4	5	1	0
---	---	---	---	---	---	---	---

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

7	6	5	3	4	2	1	0
---	---	---	---	---	---	---	---

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

7	6	5	4	3	2	1	0
---	---	---	---	---	---	---	---

– ...

## analisi della complessità efficiente

**INVERTI-ARRAY-CON-SCAMBI (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-CON-SCAMBI**( $A$ ) ha complessità  $\Theta(n)$  in quanto esegue per  $\Theta(n)$  volte il blocco delle istruzioni che consiste nell'esecuzione di una procedura  $\Theta(1)$