Complessità

Fondamenti di Informatica I
Corso di laurea in Ingegneria Informatica e Automatica
Sapienza Università di Roma

Domenico Lembo, Paolo Liberatore,
Alberto Marchetti Spaccamela, Marco Schaerf

Efficienza dei programmi

- Efficienza: fare le cose con poche risorse
- Risorse di interesse tempo di calcolo e spazio (memoria richiesta)
- In questo focalizziamo l'attenzione sul tempo. Perché?
- Nella grande maggioranza dei casi
  - lo spazio richiesto da un programma dipende dalla memoria necessaria per memorizzare l'input e i risultati
  - In particolare la memoria di lavoro (per memorizzare i risultati intermedi) è piccola (rispetto all'input)
  - Inoltre tra i diversi algoritmi / programmi non ci sono differenze significative
- Nota bene: ci sono comunque casi (anche rilevanti) in cui ci interessa minimizzare lo spazio usato. Nel seguito considereremo un caso specifico

efficienza

- efficienza: fare le cose con poche risorse
- in questo corso vediamo solo il tempo
- come misurare il tempo di esecuzione?
  - cronometriamo il programma? (apparentemente sembra la soluzione più semplice, oggettiva e facile da comunicare)
- ma il tempo di esecuzione dipende da:
  - Computer usato
  - Linguaggio di programmazione
  - Dati
  - quanti e quali dati usiamo per provare

cosa misuriamo?

Assunzione generale:

non ci interessa il tempo esatto ma una stima generale

- **Indipendente** da: computer, linguaggio, compilatore, dati di test
- **Facile** da calcolare e comunicare

Nota Bene: la stima che otterremo deve essere utile ai nostri scopi

- Deve permettere di distinguere soluzioni efficienti dalle altre
- Deve avere una rispondenza con la pratica (ad esempio non è accettabile che tutte le soluzioni siano equivalenti o, peggio, che una fra le peggiori in pratica abbia la stima migliore)

cosa misuriamo?

Assunzione generale:

non ci interessa il tempo esatto ma una stima generale del costo di esecuzione

Indipendente da: computer, linguaggio, dati

In Python…

operazioni semplici e complesse:

- Semplici
  - operazioni che coinvolgono singoli dati:

```
a=2
if b == c+2:
```

ecc.

- Complesse
  - operazioni che richiedono di guardare strutture:

```
re.search('ab*c|aab*d', 'abbbc')
if x in a:
```

ecc:

operazioni elementari

ipotesi semplificative:

- le istruzioni su dati scalari (interi, caratteri, ecc.) hanno tutte lo stesso costo (es. `a=1` ha lo stesso costo di `a==b` ma anche di `a=b*b+c-2/f`)
- le istruzioni su strutture complesse si riconducono a istruzioni su scalari

Cosa vuol dire “si riconducono” ?

operazioni su strutture complesse

Python		come viene fatto

`if x in a:`
`  print('presente')`
	⇒	`c=False`
		`for e in a:`
		`  if x==e:`
		`  c=True`
		`  break`
		`if c:`
		`  print('presente')`

il calcolatore non esegue `x in a`

invece fa un ciclo sugli elementi, confrontando ogni elemento di a con x

Assunzioni: costo di esecuzione

- solo operazioni semplici: quelle complesse le trasformiamo
- le operazioni hanno tutte lo stesso tempo di esecuzione
- **unità di tempo**
  invece di dire: ogni istruzione 1 millisecondo, o 12 nanosecondi, o 0.9 microsecondi…
  poniamo *una istruzione (semplice) = tempo 1*
  non 1 millisecondo, ma una generica unità di tempo

  equivale a dire:
  - il tempo di esecuzione si misura come numero di istruzioni semplici che vengono eseguite

Assunzioni: grandezza input

Il numero di istruzioni di un programma dipende dalla **dimensione dell'input**

- esempio: un cosa è scrivere un programma che ordina 3 numeri una cosa ben diversa è scrivere un programma che ordina un array di 1.000.000 di numeri
- esprimiamo il numero di operazioni effettuate in **funzione della dimensione dell'input,** data dalla **lunghezza dell'input** in bit (o celle di memoria) occupate
- esempio: se l'input è un array di interi, la dimensione dell'input è data dal numero di componenti dell'array
  - il tempo di esecuzione si misura come numero di istruzioni semplici che vengono eseguite **in funzione della dimensione dell'input**

programmi diversi

occorre fare qualcosa con una lista `a` e sia

len(a) la lunghezza della lista

in generale, una stessa cosa si può fare con più programmi diversi che possono avere tempi diversi

per esempio, tre programmi potrebbero distinguersi così:

1. il primo impiega $5 \times \texttt{len(a)}$
2. il secondo $100 \times \texttt{len(a)}$
3. il terzo $2^{\texttt{len(a)}}$

fanno la stessa cosa, ma con tempi diversi

confronto di tempi

len(a)	prog 1 5×len(a)	prog 2 100×len(a)	prog 3 $2^{len(a)}$
1	5	100	2
5	25	500	32
20	100	2000	1048576

Lista a corta ⇒ tempi brevi comunque

i tempi contano quando a è lunga:

- prog1 e prog2 crescono allo stesso modo
- prog3 cresce molto di più

comportamento asintotico

ci interessa come cresce il tempo quando l'input diventa grande

non ci interessa il tempo preciso

`Prog1 : 5×len(a)` e prog2: `100×len(a)` crescono in modo simile

`Prog3:` $2^{\texttt{len(a)}}$ cresce molto più velocemente

- prog1 e prog2 li consideriamo efficienti uguali
- prog3 è molto peggio

notazione O()

se:

- n = grandezza dei dati di ingresso
- tempo pari a $45 \times n^2 + 1000 \times n + 500$

si dice che il tempo è $O(n^2)$

notazione *big-O*

O-grande

Notazione O()

ci interessa come cresce il tempo quando l'input diventa grande

non ci interessa il tempo preciso

Ignoriamo costanti moltiplicative e termini di ordine inferiore

Quindi 1000 x n + 2000 = O(n); infatti

O(1000×n+2000) è come

O(1000 x n) (ignoro + 2000 e O(1000 n) = O(n)

Analogamente: 100 n^2 + 1000 n + 2000 (log n) = O(n^2); infatti

O(100 n^2)= O(n^2), O(1000 n)= O(n), O(2000 (log n))= O(log n) sono come n^2 ,n, log n, rispettivamente

1000 x n (ignoro + 2000) che è come n

Notazione O()

La notazione O() è una misura qualitativa del costo dell'algoritmo

sia n la dimensione dell'input (se a è una lista, `n=len(a)`)

esempi di costi:

- $20 \times n$
- $1000 \times n$
- $4 \times n^2$
- 2^n

primi due lineari: efficienti

terzo quadratico: un po' meno efficiente

quarto esponenziale: non efficiente

$c \times n$, $d \times n^2$ e $f \times 2^n$ li consideriamo tempi diversi

ma $c \times n$ e $d \times n$ no, ecc.

$1000 \times n + 2000$ è come $2 \times n + 1$

notazione O(): principio

- considerare il tempo in funzione della grandezza dell'ingresso
- prendere il termine che cresce di più
- ignorare costanti moltiplicative

$45 \times n^2 + 1000 \times n + 500 \Rightarrow$

$45 \times n^2 \Rightarrow$

n^2

è $O(n^2)$

notazione O: definizione

un programma ha costo (in termini di tempo) `O(f(n))` se:

- il suo tempo di esecuzione è `t(n)`

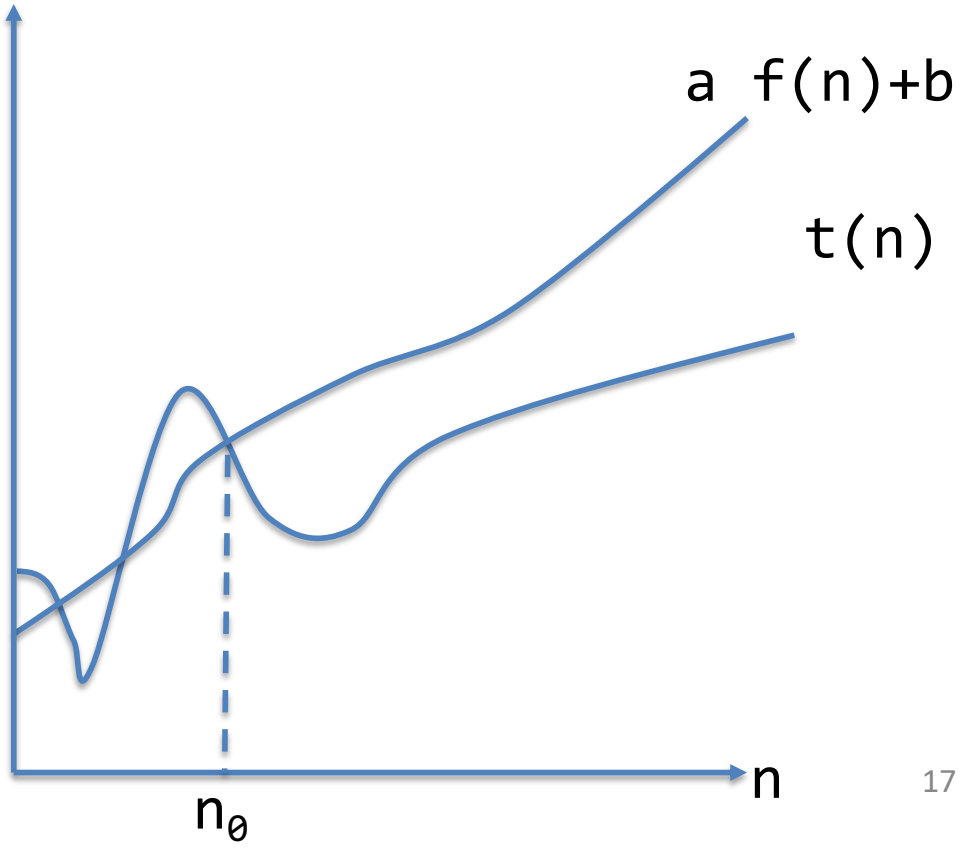
  dove n è la grandezza dell'input

- esistono opportune costanti a, b ed n_0 tali che:

$$t(n) < a\ f(n)+b$$

$$< (a+b)\ f(n)$$

per ogni $n > n_0$



notazione O: definizione

un programma ha costo (in termini di tempo) `O(f(n))` se:

- il suo tempo di esecuzione è `t(n)` (n è la grandezza dell'input)
- esistono opportune costanti a, b ed n_0 tali che:

  `t(n) < a×f(n)` per ogni n > n_0

- O(f(n)) definisce un insieme (o una classe) di funzioni

$O(f(n))=\{t: N \rightarrow N, (\text{esiste } a>0)(\text{esiste } n_0>0), (\text{tale che per ogni } n \geq n_0)\ t(n) \leq a\ f(n)\}$

- Se g(n) appartiene a O(f(n)) si dice che **"g cresce al più come f"**

- In analisi matematica due funzioni reali di variabile reale f(x) e g(x) hanno lo stesso andamento asintotico, e si scrive $f(x) \approx g(x)$, se

  $\lim_{n\to\infty} f(n)/g(n) = 1$ noi assumiamo $\lim_{n\to\infty} f(n)/g(n) = a$

- La definizione O(g(n)) usa il concetto simile di "andamento asintotico delimitato superiormente a meno di una costante"

notazione O: perché quella definizione

1. Le costanti moltiplicative e additive non contano nella notazione O; pertanto se f(n) è O(g(n)) allora anche (1000 f(n) + 1000) è O(g(n))
2. Quando si completano le funzioni, è sufficiente concentrarsi su quello che cresce di più. Ad esempio, ai fini della notazione, O ci interessa solo l'esponente più. Quindi

   f(n) = $(n^3 + 100n^2 + 1000n)$ è $O(n^3)$
3. Date due costanti a e b, a < b allora abbiamo

   n^a è $O(n^b)$ ma non vero il contrario; n^b NON è $O(n^a)$
4. Un qualunque polinomio è sempre più piccolo di una funzione esponenziale; ad esempio

   $f(n)=10n^{100}$ è $O(2^n)$ ma $g(n) = 2^n$ NON è $O(10n^{100})= O(n^{100})$
5. Analogamente una funzione logaritmica sempre più piccola di una funzione polinomiale;

- pertanto abbiamo che $f(n) = 100n^2 \log(100n)$ 2 O(n3).

notazione O: vantaggi e svantaggi

Vantaggi: semplice da calcolare e comunicare

Svantaggi: introduce approssimazioni che possono pregiudicare i risultati (almeno quando le costanti che ignoriamo sono grandi.

- Ma, nella maggioranza dei casi pratici (anche se non tutti!), in cui usiamo la notazione O le costanti nascoste non sono troppo grandi;
- in particolare a livello intuitivo, è ragionevole assumere che se f(n) è O(g(n)) allora, per ogni n sufficientemente grande
    - f(n) < 100g(n) + 1000.

Conclusione: i vantaggi sono superiori agli svantaggi e questo giustifica il trascurare i fattori costanti moltiplicativie e additivi nella notazione O().

costo delle istruzioni

con O:

- non conta se un'istruzione impiega tempo 1 o 5, ad esempio

  ```
  a=2
  a=2*b/4+35-2*c
  ```

- conta misurare il tempo in modo proporzionale alla grandezza dell'input

  ad esempio `x in a` richiede la scansione di `a`

- sono valide le assunzioni fatte sul costo delle singole istruzioni

caso migliore, medio, peggiore

Esempio: ricerca in una lista

```
c=False
for e in a:
    if x==e:
    c=True
    break
```

caso migliore

x uguale ad a[0]

caso peggiore

x non è nella lista o è alla fine

caso medio

dipende dai valori possibili di x e di a

o meglio: dalle loro probabilità

costo medio

il costo medio dipende dalle probabilità

Esempio: ricerca di `x in a`:

Se `x` e gli elementi di `a` sono valori interi qualsiasi e tutti gli elementi di `a` sono ricercabili con la stessa probabilità:

caso medio = caso peggiore

Infatti, se `x` è in posizione *i*, bisogna fare *i* confronti. Poiché la probabilità che l'intero da cercare sia in posizione *i* è sempre la stessa, allora, se `a` ha `n` elementi, la media sarà:

$1/n \times \mathrm{SUM}_{i=1..n}\ i = 1/n \times n(n+1)/2 = (n+1)/2$

(`O(n)` come nel caso peggiore).

caso peggiore

consideriamo i dati sui quali ci vuole più tempo

sulla ricerca in una lista: l'elemento non c'è

oppure è l'ultimo

in altri casi migliore, medio e peggiore coincidono:

esempio: somma degli elementi di una lista

Quando il costo di un algoritmo (o programma) è espresso usando la notazione O, si considera che l'input assuma sempre la configurazione del caso peggiore.

grandezza problemi risolubili

altro modo di vedere l'efficienza:

grandezza dell'input che un programma può risolvere in un dato tempo

Grandezza massima dell'input

	1 secondo	1 minuto (=60 secondi)	1 ora (=3600 secondi)	1 giorno (=86400 secondi)
$100 \times n$	10	600	36000	864000
$10 \times n^2$	10	77	600	2939
n^3	10	39	153	442
2^n	9	15	21	26

La tabella indica il più grande input risolubile (valore di n) in un certo tempo, assumendo una operazione ogni millesimo di secondo, per algoritmi di complessità crescente ($100 \times n$, $10 \times n^2$, ecc.).

Ad esempio, se un problema ha un costo di $100 \times n$ ed in un minuto posso eseguire 60000 operazioni → n = 60000/100 = 600

Analogamente, un problema che ha un costo di $10 \times n^2$ ed in un minuto posso eseguire 60000 operazioni → n = sqrt(60000/10) = 77

Grandezza massima dell'input: commenti

	1 secondo	1 minuto (=60 secondi)	1 ora (=3600 secondi)	1 giorno (=86400 secondi)
$100 \times n$	10	600	36000	864000
$10 \times n^2$	10	77	600	2939
n^3	10	39	153	442
2^n	9	15	21	26

all'inizio numeri simili (prima colonna), ma per un problema grande, ad esempio, 200:

- il primo algoritmo ce la fa in meno di un minuto
- al secondo serve più di un minuto
- al terzo più di un'ora
- il quarto non riesce a risolverlo nemmeno in un giorno intero

Costi di esecuzione: nomenclatura

grandezza dell'input: n

costo costante: $O(1)$ (cioè la complessità non dipende dalla dimensione n dei dati di ingresso)

costo logaritmico: $O(\log n)$

costo lineare: $O(n)$ (esempio $100 \times n + 2010$)

Costo pseudolineare: $O(n \log n)$

costo quadratico: $O(n^2)$ (esempio $10 \times n^2 + 500 \times n + 9$)

costo cubico: $O(n^3)$

costo polinomiale: $O(n^c)$, con $c > 0$

costo esponenziale: $O(2^n)$

valutazione qualitativa dei costi

polinomiale è meglio di esponenziale

- se polinomiale, è meglio un grado basso
- es. $O(n^2)$ è meglio di $O(n^3)$

Nota: *in base alla definizione, se un algoritmo (o programma) ha un costo, ad esempio $O(n)$, ovviamente esso ha anche un costo $O(n^2)$. E' chiaro che in questo caso $O(n)$ fornisce una indicazione sull'andamento asintotico del costo più precisa rispetto a $O(n^2)$. In generale, fra le varie funzioni $f(n)$ tali che il costo del programma è $O(f(n))$ si cerca la "più piccola".*

Istruzione dominante

Intuitivamente:

è una qualsiasi delle istruzioni che vengono eseguite più volte

sono quelle dei cicli “più interni”.

Esempio: Somma degli elementi in una lista

```
a=[3,9,-1,4,3]
s=0
for x in a:
   s=s+x
print(s)
```

istruzione dominante: numero di esecuzioni

programma	numero di esecuzioni
`a=[3,9,-1,4,3]`	1
`s=0`	1
`for x in a:`	
`  s=s+x`	5
`print(s)`	1

istruzione eseguita più volte: `s=s+x`

è l'istruzione dominante di questo programma

con lista qualsiasi?

Costo, con lista generica

Non siamo interessati al tempo di esecuzione con input specifico

Vogliamo sapere quanto cresce il tempo quando cresce l'input

⇒ valutazione con lista di lunghezza arbitraria

programma	numero di esecuzioni
`s=0`	1
`for x in a:`	
`  s=s+x`	n
`print(s)`	1

lista a di lunghezza n

Istruzione dominante e notazione O-grande

programma	numero di esecuzioni
`s=0`	1
`for x in a:`	
`  s=s+x`	n
`print(s)`	1

lista lunga n

⇓

l'istruzione `s=s+x` viene eseguita n volte

osservazione: il costo dell'intero programma è $O(n)$

costo = numero di esecuzioni dell'istruzione dominante

Calcolo del valore di un polinomio

$$p = c_{n-1}\, x^{n-1} + c_{n-2}\, x^{n-2} + \ldots + c_1\, x^1 + c_0\, x^0$$

- dati: coefficienti c_i e x
- lista con i coefficienti in ordine inverso
- il programma deve sommare $c_e\, x^e$
- assumiamo di non avere a disposizione l'istruzione x**e per il calcolo dell'elevamento a potenza

  calcoliamo x^e con un ciclo:

```
pt=1
for i in range(0,e):
    pt=pt*x
```

va ripetuto per ogni `e in range(0,len(coefficienti))`, dove `coefficienti` è la lista di tutti i c_i

valore polinomio: programma

```
v=0 #valore del polinomio
for e in range(0,len(coefficienti)):
    pt=1
    for i in range(0,e):
        pt=pt*x
#ora pt=x elevato alla e
    v=v+coefficienti[e]*pt    #somma termine
print('valore del polinomio:', v)
```

Nota: Qui viene nascosto il fatto che `coefficienti[e]` ha a sua volta un costo lineare, che andrebbe considerato. Sarebbe in realtà meglio sostituire il ciclo esterno con un ciclo `for c in coefficienti` e poi incrementare `e` ad ogni passo (o, usare come ciclo `for e,c in enumerate(coefficienti)`). Come vedremo nella prossima slide, la complessità comunque non cambia, dato che allo stesso livello di nidificazione in cui viene usato `coefficienti[e]` c'è il ciclo di `i`. Per semplicità invece assumiamo che `len(coefficienti)` abbia costo costante

Valore polinomio: alternative

```
v=0
e = 0
for c in coefficienti:
    pt=1
    for i in range(0,e):
        pt=pt*x
    e=e+1
    v=v+c*pt
print('valore del polinomio:', v)
```

```
v=0
for e,c in enumerate(coefficienti):
    pt=1
    for i in range(0,e):
        pt=pt*x
    v=v+c*pt
print('valore del polinomio:', v)
```

calcolo della potenza con ciclo: costi

programma	numero di esecuzioni
`v=0`	1
`for e in range(0,len(coefficienti)):`	
`  pt=1`	n
`  for i in range(0,e):`	
`    pt=pt*x`	???
`  v=v+coefficienti[e]*pt`	n
`print ('valore del polinomio:', v)`	1

dove n = numero coefficienti

- ciclo `for e...` eseguito n volte
- a ogni iterazione c'è un ciclo `for i...` di e iterazioni

calcolo della potenza con ciclo: ciclo interno

programma		numero di esecuzioni
`v=0`		1
`for e in range(0,len(coefficienti)):`		
`  pt=1`		n
`  for i in range(0,e):`		
`    pt=pt*x`		???
`  v=v+coefficienti[e]*pt`		n
`print ('valore del polinomio:', v)`		1

ciclo su `i`: con `e=0` → zero iterazioni,
con e=1 → una iterazione,
con e=2 → due iterazioni, …
con e=n-1 → n-1 iterazioni, …

Istruzione eseguita $1+2+3+\ldots+(n-2)+(n-1) = (n-1)\times n/2 = 1/2 \times n^2 - 1/2 \times n$

calcolo della potenza con ciclo: istruzione dominante

programma		numero di esecuzioni
`v=0`		1
`for e in range(0,len(coefficienti)):`		
`  pt=1`		n
`  for i in range(0,e):`		
`    pt=pt*x`		???
`  v=v+coefficienti[e]*pt`		n
`print ('valore del polinomio:', v)`		1

istruzione dominante: pt=pt*x

eseguita n^2 volte (a parte la costante 0.5)

costo $O(n^2)$

algoritmo migliore

sempre assumendo di non usare x**e per il calcolo della potenza

a ogni passo ricordo x^e

al passo successivo basta fare $x*x^e$ per ottenere x^{e+1}

```
v=0
pt=1
for c in coefficienti:
    v=v+c*pt
    pt=pt*x
print('valore del polinomio:', v)
```

costo di esecuzione? istruzione dominante?

algoritmo migliore: valutazione del costo

programma	numero di esecuzioni
`v=0`	1
`pt=1`	1
`for e in coefficienti:`	
`  v=v+e*pt`	n
`  pt=pt*x`	n
`print('valore del polinomio:', v)`	1

istruzioni dominanti: `v=v+c*pt` e `pt=pt*x`

eseguite n volte

costo `O(n)`! (non era $O(n^2)$?)

Osservazione

stesso problema

due programmi:

- il primo ha costo $O(n^2)$
- Il secondo ha costo $O(n)$

Il secondo programma è una soluzione migliore dello stesso problema

Istruzione dominante: in concreto

si guardano le istruzioni dentro i cicli più interni

si vede quante volte vengono eseguite

sempre in funzione della grandezza del l'input

questo dice il costo del programma in notazione O-grande

Esempio: ricerca binaria

1.4.1 Ricerca binaria

Consideriamo ora il seguente algoritmo di Ricerca binaria.

```
1. /*
2.   restituisce indice della posizione di k
3.   oppure -1 se k non presente; assume n > 0
4. */
5. int ricercaBinaria(int a[], int n, int k) {
6.   int primo = 0, ultimo = n - 1, media;
7.   while(primo <= ultimo) {
8.     media = (primo + ultimo) / 2;
9.     if(a[media] == k) return media;
10.     else if(a[media] < k) primo = media + 1;
11.     else ultimo = media - 1;
12.    }
13.   return -1;
14. }
```

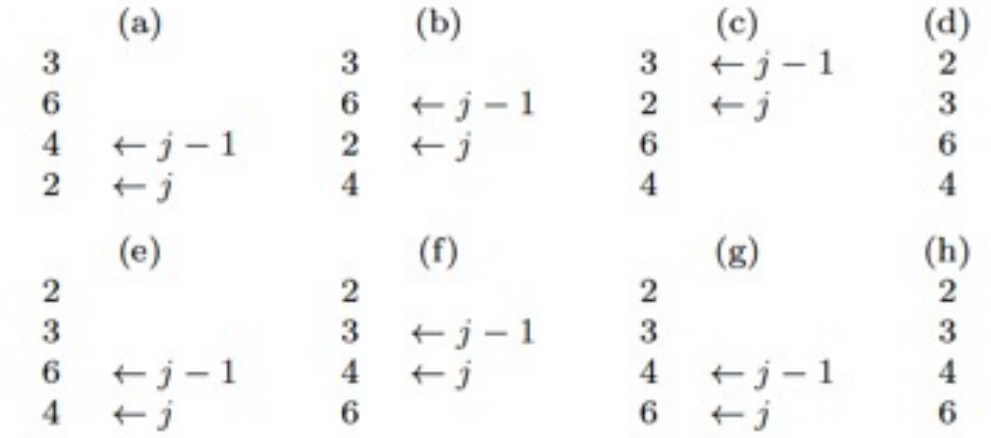
- Dimensione input: n,numero di elementi array a
- Istruzioni dominanti: 7,8,9 (interno del ciclo)
- Costo programma è O(f(n)) dove f(n) denota numero esecuzioni delle istruzioni 7 (o 8 o 9) nel caso peggiore con input di dimensione n
- No. Ripetizioni del ciclo nel caso peggiore è $O(\log_2 n)$; quando cerco ad esempio il primo o l'ultimo elemento dell'array

Esempio: test primalità

```
37.  int primo(int n) {
38.   /*
39.   restituisce 1 se n è primo
40.   oppure un divisore <= sqrt(n)
41.   assume n > 1
41. */
43.   int d;
44.   if(n <= 3) return 1;              /* ritorna se n è 2 o 3 */
45.   if(!(n % 2)) return 2;            /* ritorna se n è pari */
46.   for(d = 3; d <= sqrt(n); d += 2)  /* inutile testare numeri pari */
47.     if(!(n % d))
48.       return d;                     /* n è divisibile per d */
49.   return 1;                         /* n è primo */
50. }
```

- Dimensione input: z=numero di bit di n; il numero di bit di n è pari a $z = log_2 n$
- Il programma verifica tutti i numeri dispari minori di $\sqrt{n}$ e questi sono $\sqrt{n}/2 \approx 2^{(z-1)/2}$
- Quindi il costo del programma è **esponenziale** nella dimensione dell'input
- Nota: in generale quando abbiamo problemi i cui dati sono numeri assumiamo che i singoli dati numerici possano essere rappresentati in una (o due) locazioni di memoria; in questo caso invece siamo interessati al problema per numeri molto grandi (ad es. in crittografia usiamo numeri primi interi di 1000 o anche 2000 bit)

Ordinamento a bolle (bubblesort)



(a)		(b)		(c)		(d)
3		3		3	$\leftarrow j-1$	2
6		6	$\leftarrow j-1$	2	$\leftarrow j$	3
4	$\leftarrow j-1$	2	$\leftarrow j$	6		6
2	$\leftarrow j$	4		4		4

(e)		(f)		(g)		(h)
2		2		2		2
3		3	$\leftarrow j-1$	3		3
6	$\leftarrow j-1$	4	$\leftarrow j$	4	$\leftarrow j-1$	4
4	$\leftarrow j$	6		6	$\leftarrow j$	6

Figura 1.3 Esecuzione dell'algoritmo di ordinamento a bolle

Esempio bubblesort

(d) fine prima fase

(f) fine seconda fase

(h) fine terza fase

- NOTA: non sempre sono necessarie n - 1 fasi, ma può accadere che il vettore sia stato ordinato prima che l'ultima fase sia completata. In particolare, osserviamo che
- se durante una fase non avviene nessuno scambio allora il vettore ordinato;
- se durante una fase avviene almeno uno scambio allora non possiamo sapere se il vettore sia stato ordinato o meno.

Ordinamento a bolle (bubblesort)

NOTA: non sempre sono necessarie n - 1 fasi, ma può accadere che il vettore sia stato ordinato prima che l'ultima fase sia completata. In particolare, osserviamo che

- se durante una fase non avviene nessuno scambio allora il vettore ordinato;
- se durante una fase avviene almeno uno scambio allora non possiamo sapere se il vettore sia stato ordinato o meno.

CASO PEGGIORE: vettore ordinato in senso decrescente. In questo caso abbiamo

- Operazione dominante: confronto fra due componenti array
- Prima fase richiede n-1 confronti, seconda fase n-2, terza n-3…
- In totale il numero di confronti è

  (n-1) + (n-2) + (n-3) + ….+ 2 +1= $\sum_{i=1}^{n-1} i$ = n(n-1)/2 = $O(n^2)$

Esempio: numeri di Fibonacci e complessità programmi ricorsivi

Consideriamo il seguente frammento di programma ricorsivo

```
def fib(n)
if(n==1 || n==2) fib = 1;
else{ fib(n) = fib(n-1)+ fib(n-2)}
```

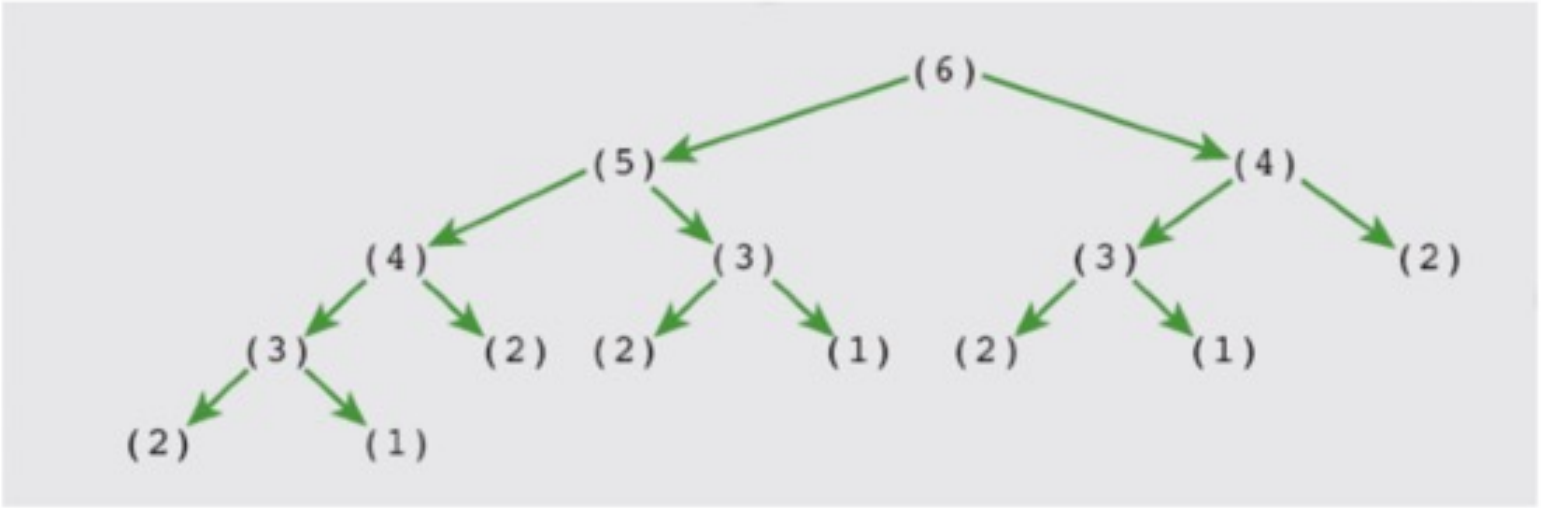
- Nota che, se $n \geq 2$ allora il frammento richiede un numero costante di operazioni che si sommano al costo di due chiamate ricorsive
- Quindi possiamo dedurre che il costo del programma è O(# ch.ric.), dove # ch.ric denota il numero di chiamate ricorsive.

Esempio: numeri di Fibonacci e complessità programmi ricorsivi

Consideriamo il seguente frammento di programma ricorsivo (n ≥1)

```
def fib(n)
if(n==1 || n==2) fib = 1;
else{ fib(n) = fib(n-1)+ fib(n-2)}
```

- Nota che, il frammento richiede un numero costante di operazioni che, nel caso che sia n≥ 2, si sommano al costo di due chiamate ricorsive
- Quindi possiamo dedurre che il costo del programma è O(# ch.ric.), dove # ch.ric denota il numero di chiamate ricorsive. Es. con n=6 abbiamo

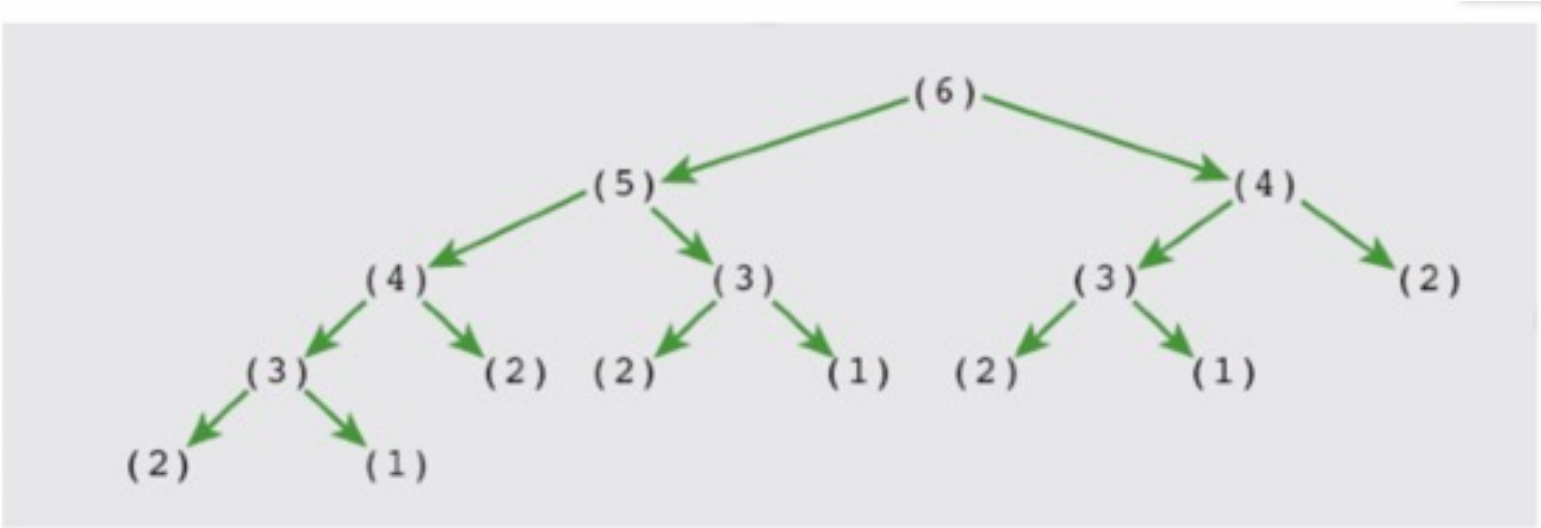


- Con n=6 abbiamo 15 chiamate ricorsive; con n=100????

Esempio: numeri di Fibonacci e complessità programmi ricorsivi

```
def fib(n)
if(n==0 || n==1) fib = 1;
else{ fib(n) = fib(n-1)+ fib(n-2)}
```

- quando n=6 abbiamo
- un'attivazione di fib(6) e una di fib(5);
- due attivazioni di fib(4) e 3 di fib(3)
- 5 attivadezioni di fib(2) e 3 di fib)1)
- Nel caso generale???



numeri di Fibonacci e complessità programmi ricorsivi

```
def fib(n)
if(n==1 || n==2) fib = 1;
else{ fib(n) = fib(n-1)+ fib(n-2)}
```

- Sia T(n) numero attivazioni ricorsive con input n. Possiamo scrivere la seguente equazione di ricorrenza

  $T(n) = 1$ se $n = 1, 2$ e $T(n) = T(n - 1) + T(n -2) + 1$ se $n \geq 3$

- in base alla definizione che per ogni n abbiamo: $T(n-1) \geq T(n-2)$
- e quindi possiamo scrivere

  $T(n) = 1$ se $n = 1; 2$ e $T(n) \leq 1+ 2T(n -1)$ se $n \geq 3$

- srotolando l'equazione precedente otteniamo

$$T(n) \leq 1 + 2T(n - 1)$$
$$\leq 1 + 2(2T(n - 2) + 1) = (1 + 2) + 4T(n - 2)$$
$$\leq (1 + 2) + 4(2T(n-3) + 1) = (1+ 2 + 4) + 8T(n -3)$$
$$\leq (1+2 + 4) + 8(2T(n - 4) + 1) = (1+2 + 4 + 8) + 16T(n -4)$$
$$\ldots\ldots$$
$$\leq (1+2 + 4 + 8 + 16 \ldots . 2^{k-1}) + 2^k T(n - k)$$

Quando ci fermiamo???

numeri di Fibonacci e complessità programmi ricorsivi

- Sia T(n) numero attivazioni ricorsive con input n. Possiamo scrivere la seguente equazione di ricorrenza

  $T(n) = 1$ se $n = 1, 2$ e $T(n) = T(n - 1) + T(n - 2) + 1$ se $n \geq 3$

- in base alla definizione che per ogni n abbiamo: $T(n-1) > T(n-2)$
- e quindi possiamo scrivere

  $T(n) = 1$ se $n = 1; 2$ e $T(n) \leq 1 + 2T(n - 1)$ se $n \geq 3$

- srotolando l'equazione precedente otteniamo

$$T(n) \leq 1 + 2T(n - 1)$$
$$\leq 1 + 2(2T(n - 2) + 1) = (1 + 2) + 4T(n - 2)$$
$$\leq (1+2) + 4(2T(n-3) + 1) = (1+2 + 4) + 8T(n -3)$$
$$\leq (1+2 + 4) + 8(2T(n - 4) +1) = (1+2 + 4 + 8) + 16T(n -4)$$
$$\ldots\ldots$$
$$\leq (2 + 4 + 8 + 16 \ldots . 2^k) + 2^k T(n - k)$$

ci fermiamo quando $n- k=2$ (cioè $k=n-2$) e otteniamo

$$\leq (1 +2+ 4 + 8 + 16 \ldots . 2^{n-2}) = 2^n-1$$

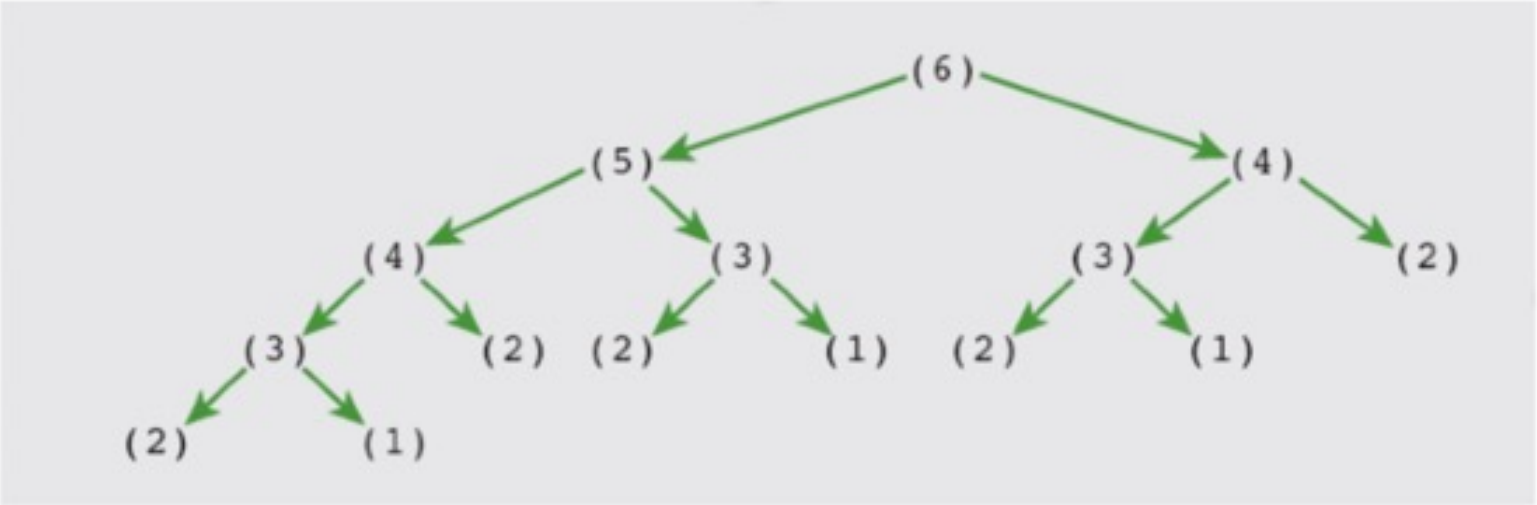
dato che $2^{n+1} = 2(2^n)$. Il costo è $O(2^n)$ esponenziale in n

Esempio: numeri di Fibonacci e complessità programmi

Consideriamo il seguente frammento di programma ricorsivo (n ≥1)

```
def fib(n)
if(n==1 || n==2) fib = 1;
else{ fib(n) = fib(n-1)+ fib(n-2)}
```

- Nota che, il frammento richiede un numero costante di operazioni che, nel caso che sia n≥ 2, si sommano al costo di due chiamate ricorsive
- Quindi possiamo dedurre che il costo del programma è O(# ch.ric.), dove # ch.ric denota il numero di chiamate ricorsive. Es. con n=6 abbiamo



- Con n=6 abbiamo 15 chiamate ricorsive; con n=100????

Esempio: numeri di Fibonacci e complessità programmi ricorsivi

Consideriamo il seguente frammento di programma iterativo (n ≥1)

```
if(N==2 || N==1) fib = 1;
else{
fib1=1; fib2=1;
i=1;
while(i<N){
fib=fib1+fib2;
i++;
fib2=fib1; fib1=fib;
}
}
printf("Il valore di fibonacci di %d e‘: %d\n",N,fib)
```

- Il costo dell’algoritmo è O(k) dove k è il numero di iterazioni del ciclo
- È facile vedere che il numero di iterazoni k = n

problemi e programmi

uno stesso problema si può risolvere con più programmi

alcuni possono essere più veloci di altri, per es:

- per un problema esiste un programma quadratico ($O(n^2)$) e uno lineare ($O(n)$)
- per un altro problema esiste un programma esponenziale ($O(2^n)$) e uno cubico ($O(n^3)$)

ovviamente, si prende il programma migliore

complessità e costo

Programma:

notazione O() esprime una delimitazione superiore al costo di un programma

costo di un programma (= quante istruzioni vengono eseguite)

Problema: siamo interessati al migliore programma

complessità di un problema (= costo del suo programma migliore)

complessità di un problema

- Introduciamo la notazione $\Omega()$ per esprimere una delimitazione inferiore al costo di un problema
- Possiamo definire $\Omega()$ come il costo del migliore programma di soluzione?

Non è sufficiente: non consideriamo sviluppi futuri

- ci chiediamo quale posa essere la complessità intrinsica di un problema (il costo del miglior algoritmo possibile fra quelli noti e quelli - al momento ignoti – che potrebbero essere definiti)

notazione O(): definizione (già vista)

un programma ha costo (in termini di tempo) `O(f(n))` se:

- il suo tempo di esecuzione è `t(n)`

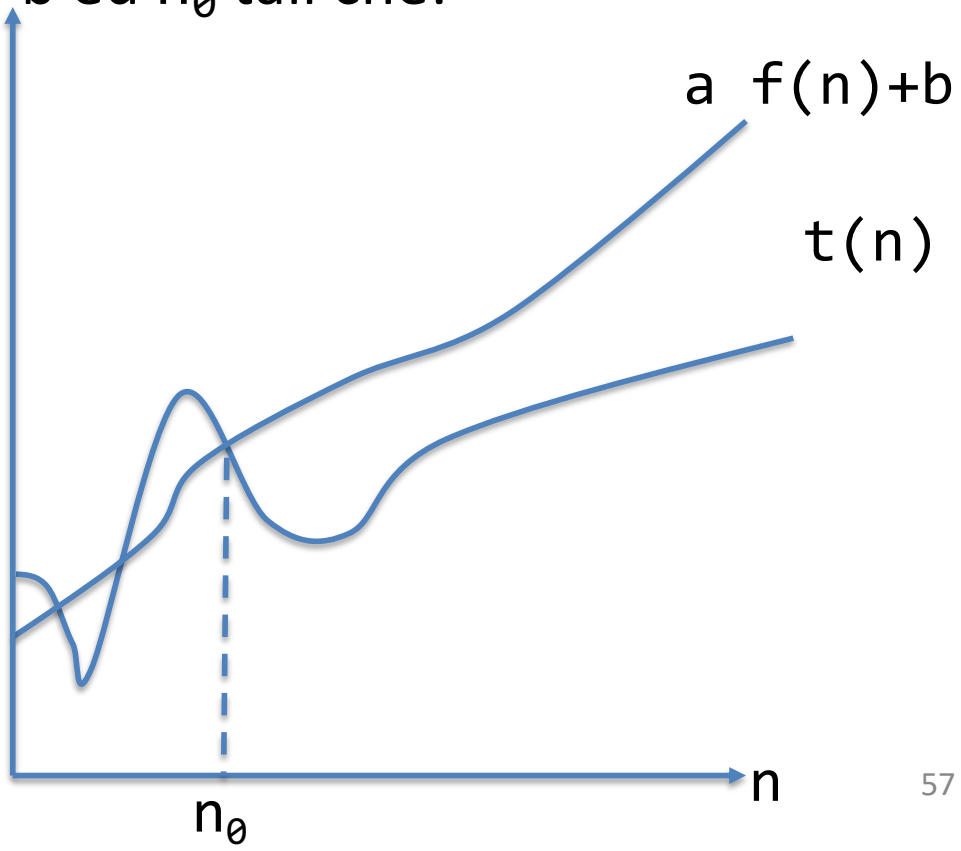
  dove n è la grandezza dell'input

- esistono opportune costanti a, b ed n_0 tali che:

$$t(n) < a\ f(n)+b$$

$$< (a+b)\ f(n)$$

per ogni $n > n_0$



notazione O: definizione

un programma ha costo (in termini di tempo) `O(f(n))` se:

- il suo tempo di esecuzione è `t(n)` (n è la grandezza dell'input)
- esistono opportune costanti a, b ed n_0 tali che:

  `t(n) < a×f(n) + b` per ogni n > n_0

- O(f(n)) definisce un insieme (o una classe) di funzioni

O(f(n))={t: N→N,(esiste a>0)(esiste n_0>0),(tale che per ogni n≥n_0) t(n)≤a f(n)}

- Se g(n) appartiene a O(f(n)) si dice che **"g cresce al più come f"**

- In analisi matematica due funzioni reali di variabile reale f(x) e g(x) hanno lo stesso andamento asintotico, e si scrive f(x) ≈ g(x), se

  $\lim_{n\to\infty}$ f(n) /g(n) =1 noi assumiamo $\lim_{n\to\infty}$ f(n) /g(n) = a

- La definizione O(g(n)) usa il concetto simile di "andamento asintotico delimitato superiormente a meno di una costante"

Notazione Ω: definizione

La notazione O() esprime una **delimitazione superiore**

- Se f(n) appartiene a O(g(n)) si dice che **"f cresce al più come g,"**

La notazione Ω() esprime una **delimitazione inferiore.** In particolare

Un programma ha costo (in termini di tempo) Ω(f(n)) se:

- il suo tempo di esecuzione è t(n) (n è la grandezza dell'input)
- esistono opportune costanti a, a≥0, (a <1 possibile) ed n_0 tali che:

  $t(n) \geq a \times f(n)$ per ogni $n > n_0$

Nota: differenza fra O() Ω (): in O() chiediamo ≤ in Ω() richiediamo ≥

Come nel caso di O() Ω(f(n)) definisce un insieme di funzioni

- Ω (f(n))={t: N→N (esiste a>0)(esiste n_0>0),(tale che per ogni n≥n_0) t(n) ≥ a f(n)}.
- Se f(n) appartiene a Ω(g(n)) si dice che **"f cresce almeno come g,"**

Notazione Ω

La notazione Ω() esprime una **delimitazione inferiore** utile per esprimere il costo intrinseco di un programma/problema.

- Ricorda: Se f(n) appartiene a O(g(n)) si dice che **"f cresce al più come g,"**

In particolare per un programma diciamo

Un programma ha costo (in termini di tempo) $\Omega(f(n))$ se:

- il suo tempo di esecuzione è $t(n)$ (n è la grandezza dell'input)
- esistono opportune costanti a, b ed n_0 tali che:

$$t(n) \geq a \times f(n) \quad \text{per ogni } n > n_0$$

Nota: differenza fra O() Ω (): in O() chiediamo ≤ in Ω() richiediamo ≥

Come nel caso di O() Ω(f(n)) definisce un insieme di funzioni

Ω (f(n))={t: N→N (esiste a>0)(esiste n_0>0),(t.c. per ogni n≥n_0) t(n) ≥ a f(n)}

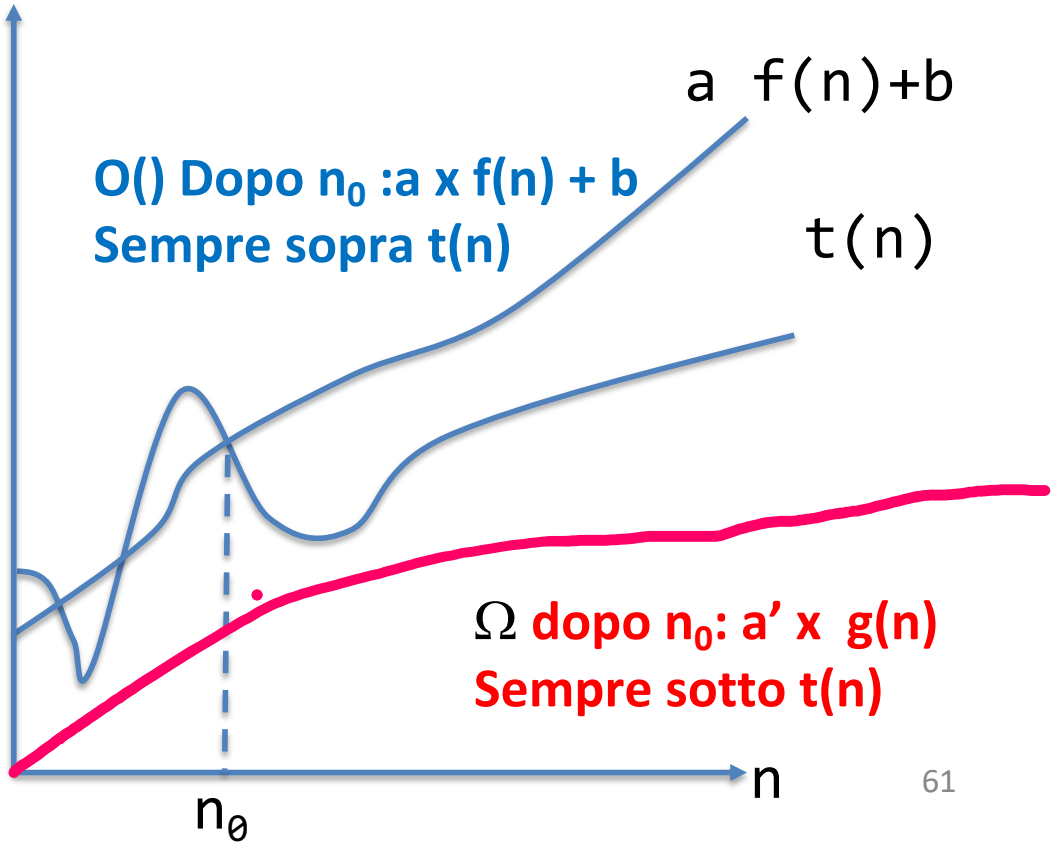
- Se f(n) appartiene a Ω(g(n)) si dice che **"f cresce almeno come g,"**

notazione O e Ω a confronto

Sia dato un programma con costo t(n)

Diciamo che un programma ha costo (in termini di tempo) Ω(f(n)) se il suo tempo di esecuzione è t(n) (dove n è la grandezza dell'input) se per ogni n > n0

- a f(n) + b sta sopra t(n)
- g(n) sta sotto t(n)



Notazione Ω

La notazione $\Omega()$ esprime una **delimitazione inferiore** utile per esprimere il costo intrinseco di un problema o equivalentemente **per esprimere una delimitazione inferiore alla complessità di un problema**

- Introduciamo la notazione $\Omega()$ per esprimere una delimitazione inferiore al costo di un problema
- Possiamo dire che se il migliore programma di soluzione ha costo $\Omega(f(n))$ allora il problema ha complessità intrinseca $\Omega(f(n))$?

Non è sufficiente: non consideriamo sviluppi futuri algoritmi che ancora non sono stati ideati

Definizione: un problema ha complessità $\Omega(f(n))$ se riusciamo a dimostrare che il costo di ogni programma (noto o che può essere ideato nel futuro) ha costo $\Omega(f(n))$

In questo modo esprimiamo la complessità intrinseca di un problema

Notazione Ω

Definizione: un problema ha complessità $\Omega(f(n))$ se dimostriamo che il costo di ogni programma (noto o che può essere ideato nel futuro) ha costo $\Omega(f(n))$

In questo modo esprimiamo la complessità intrinseca di un problema

Dimostrare limiti interessanti alla complessità di un problema è in generale molto complicato (dobbiamo immaginare limiti anche per gli algoritmi futuri). Per questo i limiti inferiori noti sono relatiivamente pochi

Due casi importanti in cui poniamo limiti sul tipo di algoritmi

Consideriamo algoritmi che usano confronti. In questo caso (vedi dispense)

1. Il costo della ricerca di un elemento in un array con n elementi ha costo $\Omega(\log(n))$
2. Il costo dell'ordinamento di un array con n elementi ha costo $\Omega(n \log(n))$

notazione θ

Sia dato un problema P: Supponiamo che

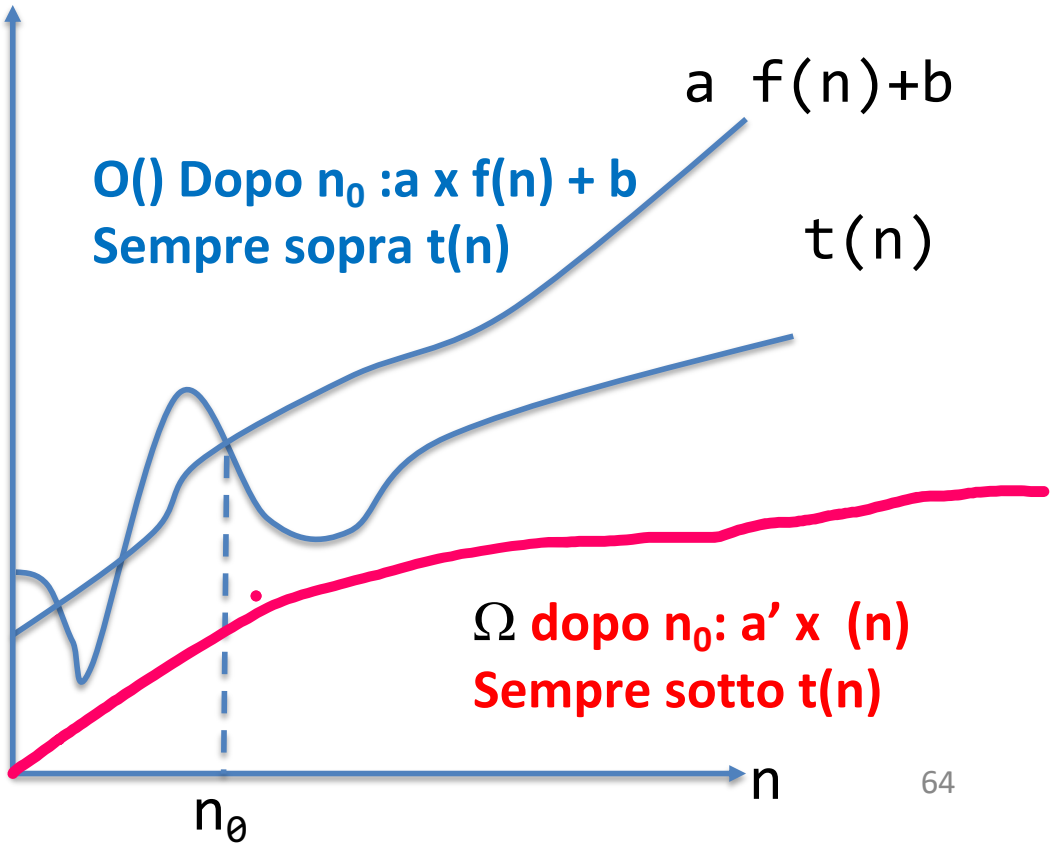
1. Conosciamo un programma di soluzione con costo t(n) = O(f(n))
2. Possiamo dimostrare che la complessità del problema è Ω(f(n))

NOTA: O(f(n)) e Ω(f(n)) con stessa funzione f(n)

1. e 2. precedenti forniscono una valutazione precisa del costo problema

Infatti la stessa funzione f(n) rappresenta Sia una delimitazione superiore che una delimitazione inferiore al costo del problema

Usiamo in questi casi la notazione Θ e diciamo che il problema ha complessità Θ(f(n))



notazione O, Ω, Θ conclusioni

Sia dato un problema

1. Se abbiamo un programma che **risolve con costo O(f(n))** allora

- Il problema ha una **delimitazione superiore alla complessità O(f(n))**

2. Se dimostriamo che **ogni algoritmo di soluzione ha costo Ω(g(n))** allora

- Il problema ha una **delimitazione inferiore alla complessità Ω(g(n))**

3. Se dimostriamo che

   3.1 esiste un programma che risolve con costo O(f(n)) e

   3.2 ogni algoritmo di soluzione ha costo Ω(f(n))

- Abbiamo definito la complessità del problema:

  f(n) rappresenta sia una delimitazione superiore che una delimitazione inferiore alla complessità e diciamo che **la complessità del problema è Θ(f(n))**

notazione O, Ω, Θ conclusioni

E' difficile dimostrare che

3.1 esiste un programma che risolve con costo $O(f(n))$

3.2 la complessità intrinseca del problema è $\Omega(f(n))$

Lo sappiamo fare in pochi casi (dimostrare delimitazioni inferiori buone valide per tutti gli algoritmi è molto complicato)

Due casi importanti in cui poniamo limiti sul tipo di algoritmi

Consideriamo algoritmi che usano confronti. In questo caso (vedi dispense)

1. Il costo della ricerca di un elemento in un array ordinato con n elementi ha costo $\Omega(\log(n))$ (algoritmo:ricerca binaria)
2. Il costo dell'ordinamento di un array con n elementi ha costo $\Omega(n \log(n))$ (algoritmo: mergesort, quicksort)

complessità e costo

programma

└ costo (= quante istruzioni vengono eseguite)

problema

└ complessità (= costo del suo programma migliore)

complessità di un problema

- si considera il programma migliore che risolve il problema
- facciamo una distinzione solo fra polinomiale ed esponenziale

P = insieme dei problemi che si risolvono in tempo polinomiale

classe P

Se non ci interessa distinguere il grado del polinomio:

P = insieme dei problemi per i quali esiste una macchina di Turing che li risolve in tempo polinomiale nella dimensione dell'input

Problemi in P

- vedere se un elemento è in una lista
- decidere il valore di una formula booleana dati i valori delle variabili
- sommare due numeri interi in precisione arbitraria
- Ordinamento di un array
- …

consideriamo sempre costo al crescere dell'input

non ha senso valutare il costo di una somma a 64 bit

Problemi non in P (forse)

- tutti i problemi indecidibili non sono in P (ovviamente)

Ci sono molti altri casi di problemi che crediamo NON siano in P ma non siamo in grado di provarlo

- decidere se una formula del calcolo proposizionale è vera per qualche assegnazione di valore delle variabili (SAT)
- colorare una carta geografica con 3 colori, in modo tale che nazioni confinanti abbiano colori diversi
- Date n città trovare il percorso più breve che passa una sola volta per tutte le città e ritorna al punto di partenza

classi di complessità

P

problemi risolubili da una macchina di Turing in tempo polinomiale

NP

problemi risolubili da una macchina di Turing non-deterministica in tempo polinomiale

Macchina di Turing non-deterministica: macchina di Turing che, diversamente da quella deterministica, è in grado di portare avanti contemporaneamente diverse elaborazioni, potenzialmente in numero illimitato

la classe NP

contiene i problemi nella stessa forma di SAT:

- esistono valori per delle variabili…
- tali che una certa condizione è vera

la condizione si può verificare in tempo polinomiale (ma ci sono molti valori da considerare)

esempio: disposizione dei cavalieri a un tavolo

EXPTIME = problemi risolubili in tempo esponenziale

$P \subset EXPTIME$

$P \subseteq NP \subseteq EXPTIME$

potrebbe essere P=NP

collocazione di NP: conseguenze

Se NP=P

anche i problemi più difficili in NP (es: SAT, ciclo Hamiltoniano) si risolvono in tempo polinomiale

Se NP=EXPTIME

i problemi più difficili di NP richiedono tempo esponenziale

molti ritengono che NP≠P BOH

premio di $1000000 per chi risolve la questione

esercizi

Consideriamo ad esempio il seguente algoritmo che trova duplicati all'interno di un array a di dimensione n.

```
ContieneDuplicati( a ) {
    trovato = false;
    for (i=0 ; i<n-1 ; i=i+1)
        { for (j=i+1 ; j<n ; j=j+1)
            { if (a[i]=a[j]) trovato=true;
            }
        }
    return trovato; }
```

esercizi

Consideriamo ad esempio il seguente algoritmo che trova duplicati all'interno di un array a di dimensione n.

```
ContieneDuplicati( a ) {
    trovato = false;
    for (i=0 ; i<n-1 ; i=i+1)
        { for (j=i+1 ; j<n ; j=j+1)
            { if (a[i]=a[j]) trovato=true;
            }
        }
    return trovato; }
```

Costo ≤ $C1 + \sum_{0 \leq i \leq n-2} \sum_{i+1 \leq j \leq n-1} C2$

C1 e C2 sono costanti; poiché siamo interessati a costo asintotico possiamo assumere C1=C2=1

esercizi

Consideriamo ad esempio il seguente algoritmo che trova duplicati all'interno di un array a di dimensione n.

```
ContieneDuplicati( a ) {
    trovato = false;
    for (i=0 ; i<n-1 ; i=i+1)
        { for (j=i+1 ; j<n ; j=j+1)
            { if (a[i]=a[j]) trovato=true;
            }
        }
    return trovato; }
```

Costo= C1 + $\sum_{0 \leq i \leq n-2} \sum_{i+1 \leq j \leq n-1}$ C2

C1 e C2 sono costanti

Esercizio: trova duplicati

Consideriamo ad esempio il seguente algoritmo che trova duplicati all'interno di un array a di dimensione n.

```
ContieneDuplicati( a ) {
    trovato = false;
    for (i=0 ; i<n-1 ; i=i+1)
        { for (j=i+1 ; j<n ; j=j+1)
            { if (a[i]=a[j]) trovato=true;
            }
        }
    return trovato; }
```

Costo= $1+ \sum_{0\leq i \leq n-2} \sum_{i+1\leq j \leq n-1} 1 = 1 + \sum_{0\leq i \leq n-2} ((n-1)-(i+1)+1) \times 1$

$\leq 1+ (n-1)(n-2)= O(n^2)$

Lo stesso risultato lo otteniamo individuando l'istruzione 'if' come istruzione dominante e contando il numero di esecuzioni della sterssa

Esercizio: array palindromo

Calcolare la complessità computazionale del seguente programma che verifica se un array a di dimensione n è palindromo, ossia se a[i] == a[n-1-i] per ogni i

```
Palindromo_Iterativo( a ) {
    i = 0;
    j = n-1;
    risposta = true;
    while (i<j) and risposta {
        if (a[i]!=a[j]) risposta=false;
        i=i+1;
        j=j-1; }
return risposta; }
```

- Caso peggiore?
- Numero di iterazioni del ciclo nel caso peggiore?

Esercizio: array palindromo

Calcolare la complessità computazionale del seguente programma che verifica se un array a di dimensione n è palindromo, ossia se a[i] == a[n-1-i] per ogni i

```
Palindromo_Iterativo( a ) {
    i = 0;
    j = n-1;
    risposta = true;
    while (i<j) and risposta {
        if (a[i]!=a[j]) risposta=false;
        i=i+1;
        j=j-1; }
return risposta; }
```

- Caso peggiore: sono eseguite $\lfloor n/2 \rfloor$ iterazioni
- Nota: è possibile migliorare il programma ?

Esercizio: array palindromo

Calcolare la complessità computazionale del seguente programma che verifica se un array a di dimensione n è palindromo, ossia se a[i] == a[n-1-i] per ogni i

```
Palindromo_Iterativo( a ) {
    i = 0;
    j = n-1;
    risposta = true;
    while (i<j) and risposta {
        if (a[i]!=a[j]) risposta=false;
        i=i+1;
        j=j-1; }
return risposta; }
```

- Caso peggiore: sono eseguite $\lfloor n/2 \rfloor$ iterazioni
- Nota: è possibile migliorare il programma ?

  Si ma il costo nel caso peggiore migliora?

Insertion sort (ordina per inserimenti)

Input: sequenza $\langle a_1, a_2, \ldots, a_n \rangle$ di numeri

Output: permutazione $\langle a'_1, a'_2, \ldots, a'_n \rangle$ dell'input tale che

$$a'_1 \leq a'_2 \leq \cdots \leq a'_n$$

Ordinamento in senso crescente

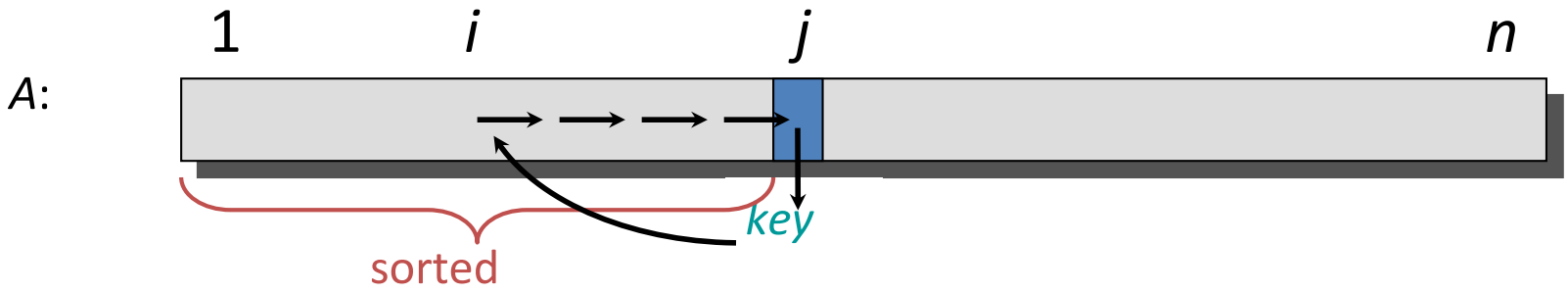
Input: 8 2 4 9 3 6

Output: 2 3 4 6 8 9

Insertion sort

"pseudocodice"

```
INSERTION-SORT (A, n)
    for j ← 2 to n
        do key ← A[j]
           i ← j – 1
           while i > 0 and A[i] > key
               do A[i+1] ← A[i]
                  i ← i – 1
           A[i+1] = key
```

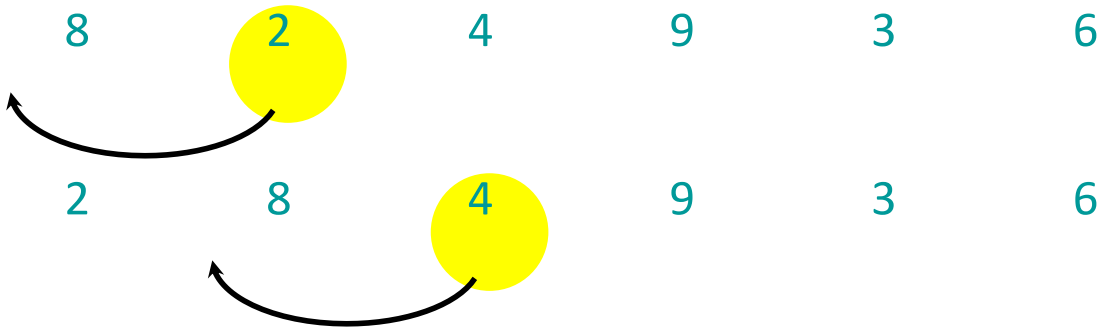


Esempio

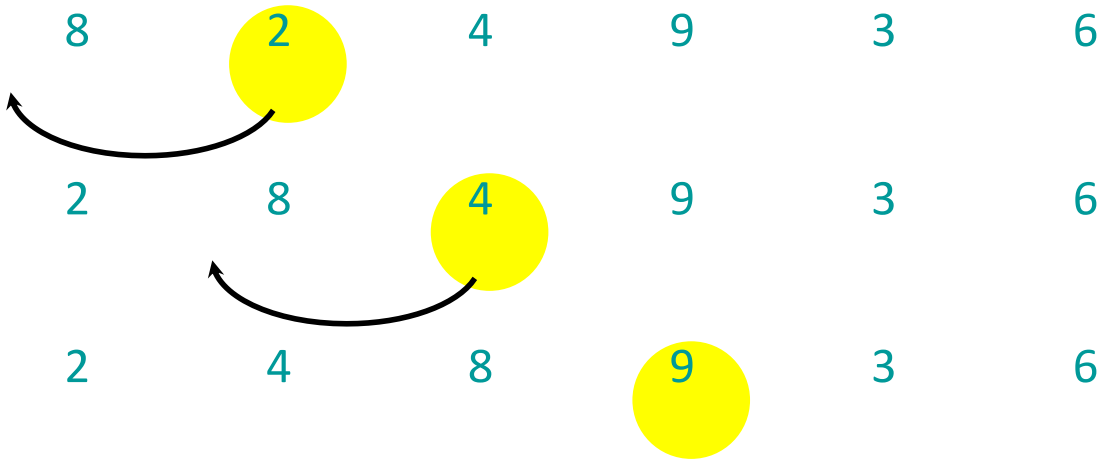
Esempio

Esempio

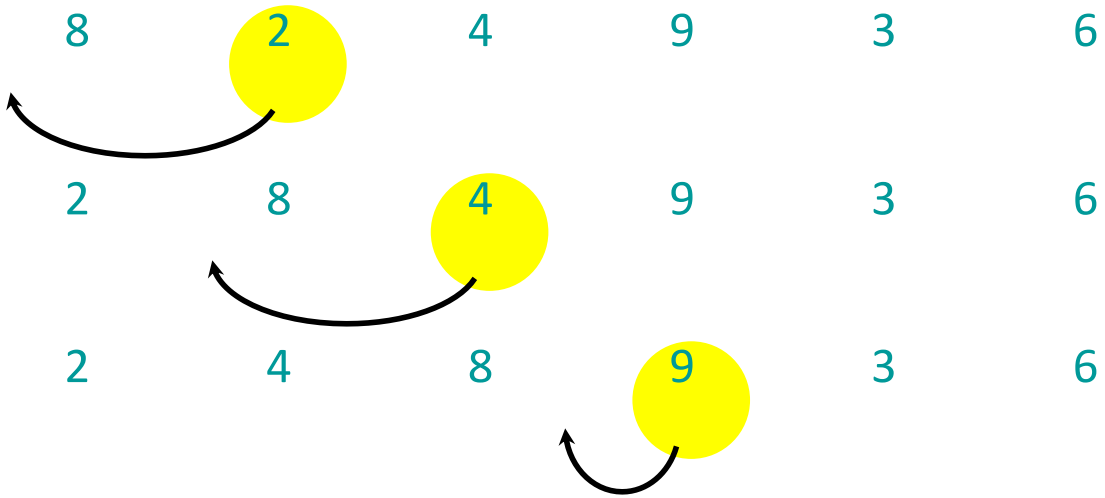
Esempio



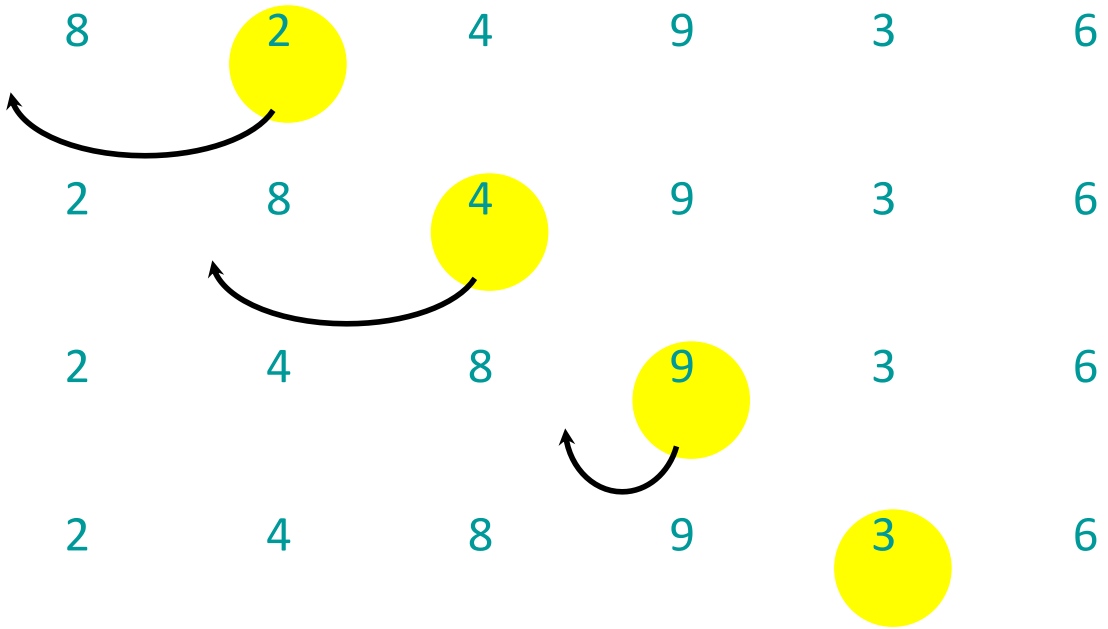
Esempio



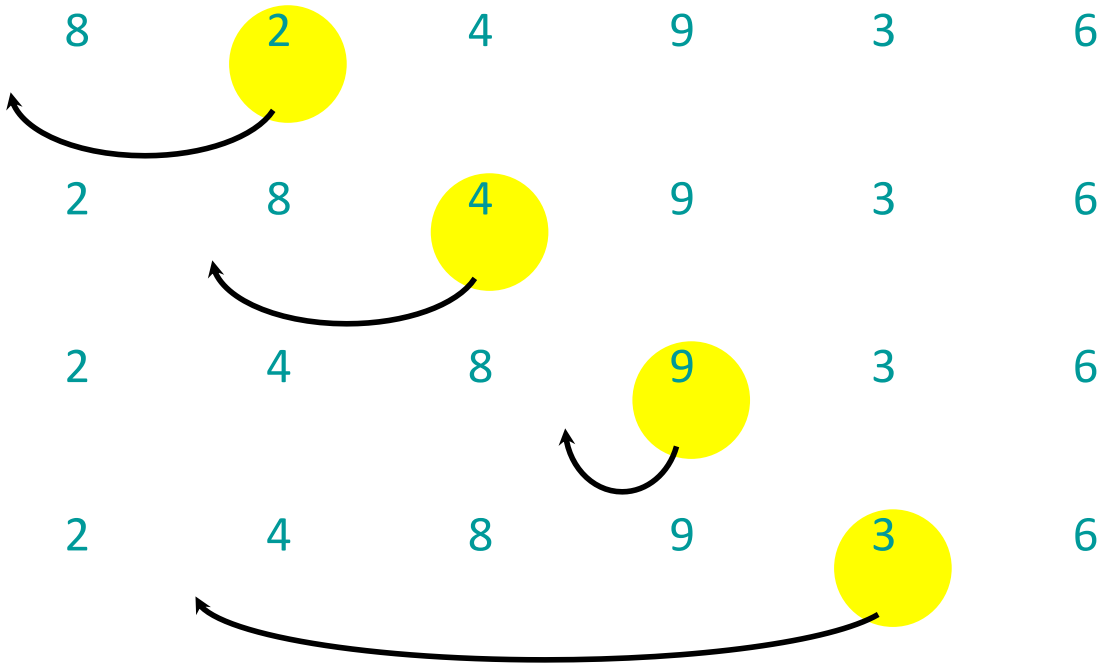
Esempio



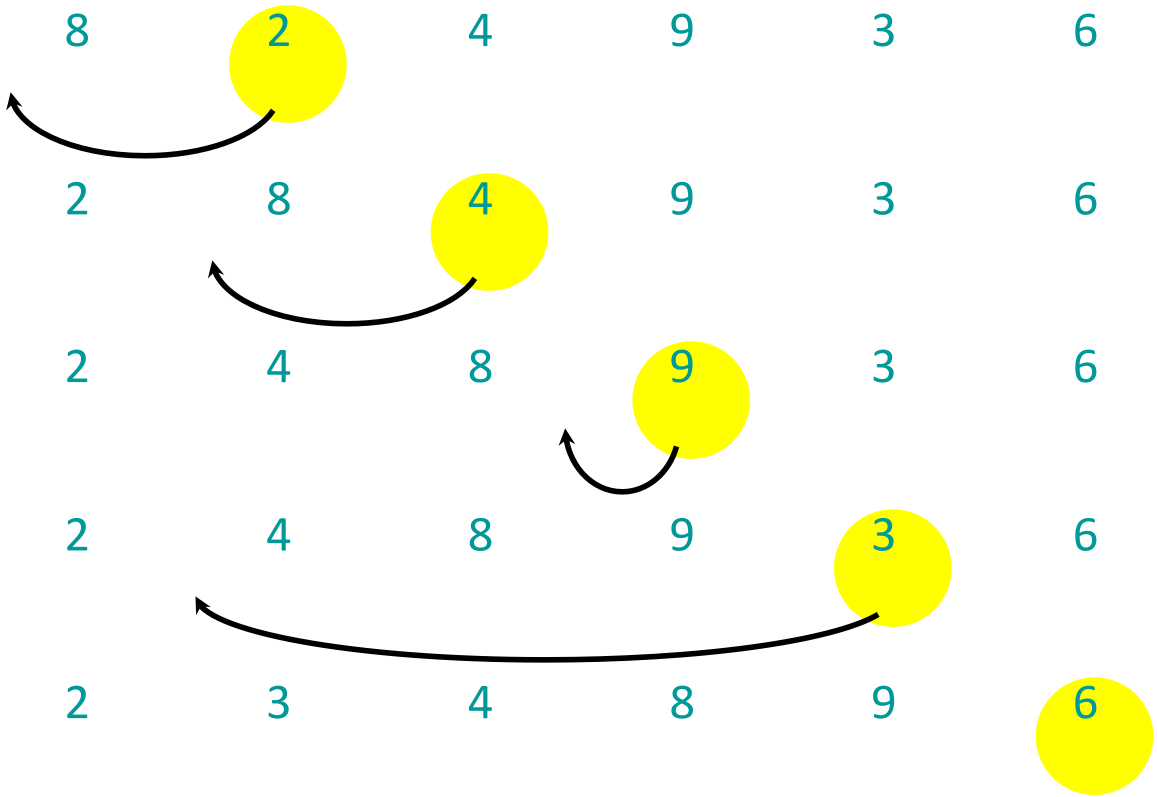
Esempio



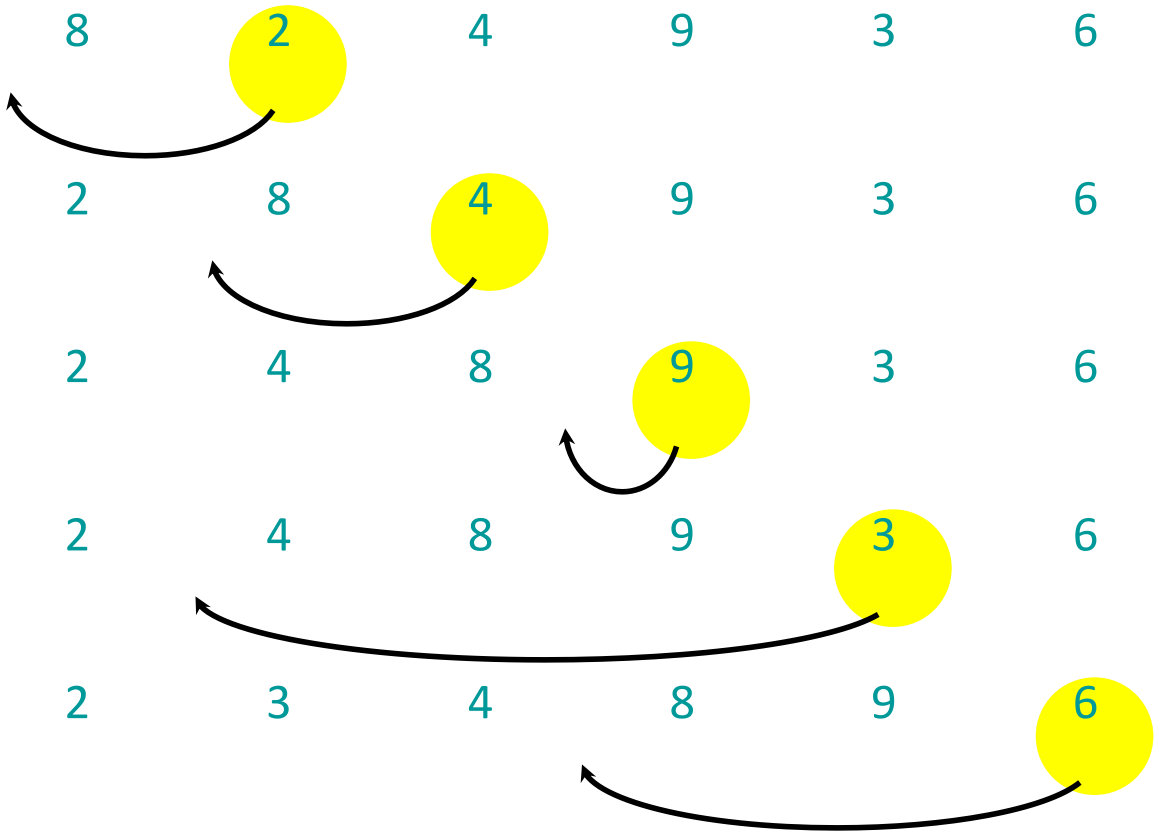
Esempio



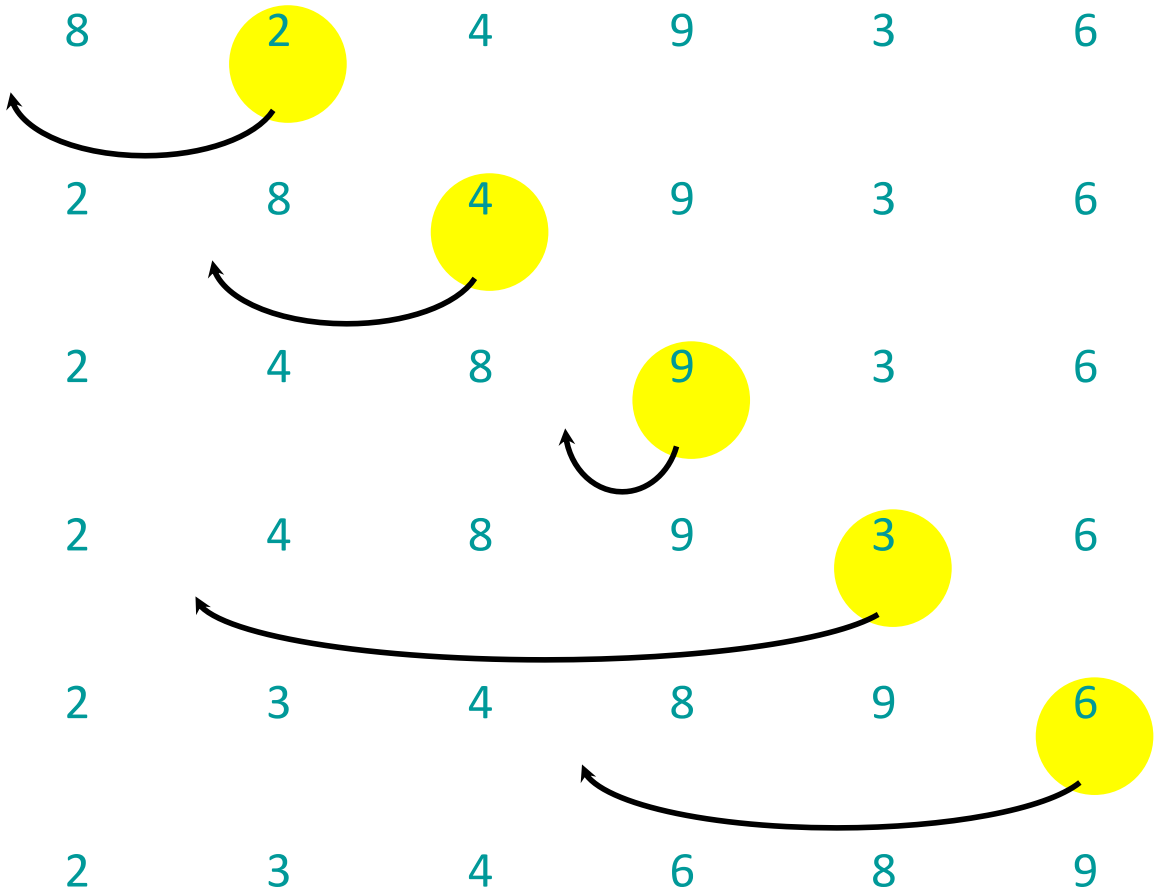
Esempio



Esempio



Esempio



Fine:
vettore ordinato

Insertion sort

"pseudocodice"

```
INSERTION-SORT (A, n)
    for j ← 2 to n
        do key ← A[j]
           i ← j – 1
           while i > 0 and A[i] > key
               do A[i+1] ← A[i]
                  i ← i – 1
           A[i+1] = key
```

Caso peggiore?

Costo nel caso peggiore?

Insertion sort

"pseudocodice"

```
INSERTION-SORT (A, n)
    for j ← 2 to n
        do key ← A[j]
           i ← j – 1
           while i > 0 and A[i] > key
               do A[i+1] ← A[i]
                  i ← i – 1
           A[i+1] = key
```

Caso peggiore: vettore ordinato in senso decrescente

Costo nel caso peggiore?