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

- efficienza: fare le cose con poche risorse
- risorse = tempo e spazio
- in questo corso vediamo solo il tempo
- come misurare il tempo di esecuzione?

cronometriamo il programma?

ma il tempo dipende da:

- Computer
- 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, 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

- 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

programmi diversi

occorre fare qualcosa con una lista `a`

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 $2^{\texttt{len(a)}}$
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

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

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

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

misura qualitativa

sia n la dimensione dell'input

se 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

se:

- n = grandezza dei dati di ingresso
- tempo pari a `45×n`^{2} `+ 1000×n + 500`

si dice che il tempo è $O(n^2)$

notazione *big-O*

O-grande

notazione O: principio

- considerare il tempo in funzione della grandezza dell'ingresso
- prendere la parte 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:

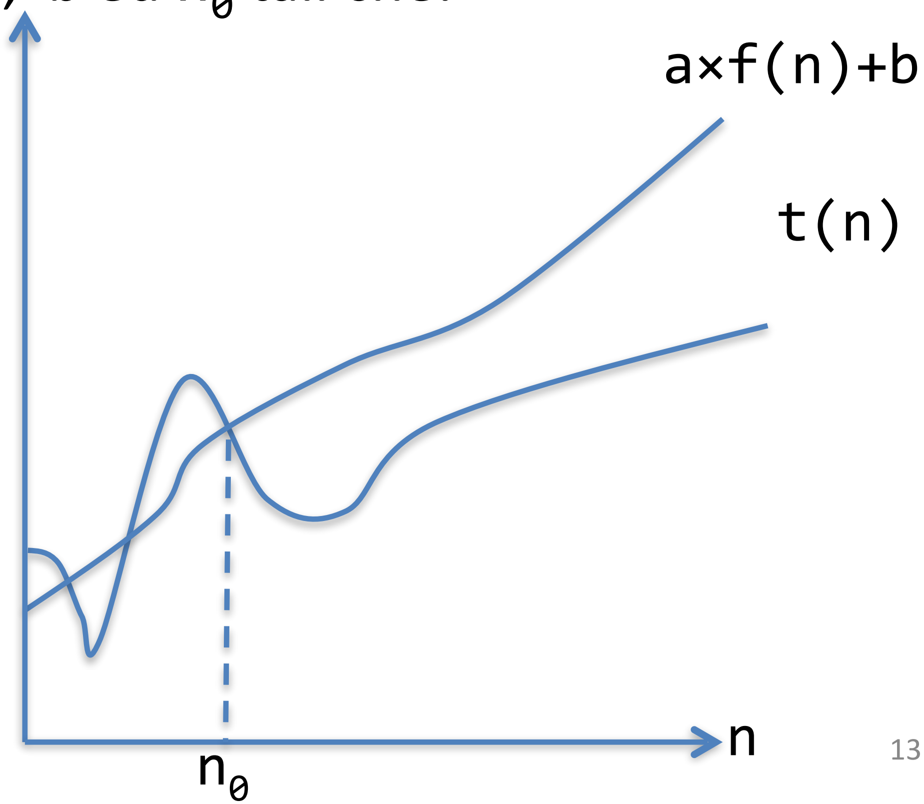
- il suo tempo di esecuzione è `t(n)`

  dove n è la grandezza dell'input

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

  `t(n) < a×f(n)+b`

  per ogni n > n_0



notazione O: perché quella definizione

sull'esempio (per $n>1$):

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

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

$1045 \times n^2 + 500 =$

$a \times n^2 + b$

costanti $a=1045$ e $b=500$

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 → $n = \text{sqrt}(60000/10) = 77$

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

$\Downarrow$

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

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

- dati: coefficienti c_i e x
- lista con i coefficienti in ordine inverso
- il programma deve sommare $c_e \times x^e$
- 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 $x \times x^e$ 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 c in coefficienti:`		
`  v=v+c*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

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

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
- …

consideriamo sempre costo al crescere dell'input

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

Problemi non in P

- tutti quelli indecidibili (ovviamente)
- decidere se il primo giocatore ha una strategia vincente nel gioco dei ciottoli
- forse: decidere se una formula booleana è vera per qualche valore della variabili (SAT)
- forse: decidere se è possibile disporre dei cavalieri a tavola, date le rivalità (ciclo Hamiltoniano)
- …

problemi non dimostrabilmente esponenziali

Per molti problemi:

- non si conoscono programmi polinomiali, ma
- non è stato dimostrato che non esistono

esempio: SAT

= decidere se una formula Booleana è vera per qualche valore delle variabili

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

premio di $1000000 per chi risolve la questione