

Appunti di programmazione lezione 7

Complessità dei programmi

- cresce asintoticamente più grande al dipendere di n

- il termine di ordine superiore può essere moltiplicato per una costante di calcolo moltiplicativa

- se $n = 10$ dimensione dell'input

$14n^2$
qualsiasi
rate
di più

- sia A un algoritmo

$T(n)$ numero di operazioni da eseguire da A con l'input

Esempi

- $T(n) = 2n^2 + \frac{n}{4} + 9$

- $T(n) = \begin{cases} 3n^2 & \text{se vera} \\ n^3 & \text{se falsa} \end{cases}$

l'ordine di grandezza

è $\cancel{2n^2}$

allora

$T(n) \in O(n^2)$

limitazione superiore

NOTAZIONI

Toto

$T(n) \in \Theta(n^2)$

$T(n)$ cresce allo stesso modo

$T(n) \in \Omega(n^2)$

$x = \text{stringa di dimensione } n = \text{len}(x)$

$c \in x$

quanto costa
questa operazione
in Python

$$T(n) = \begin{cases} i + & \text{dove } i \text{ t.c.} \\ n & x[i] == c \\ & \text{se } c \notin x \end{cases}$$

non
dipende
solo da
 n

dentro c'è una
proprietà dell'input ovvero i

ci mettiamo nell'istanza

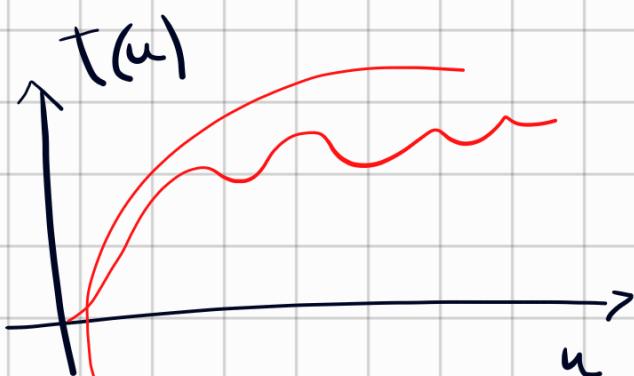
peggiore \rightarrow costo

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

semplificazione
dello funzione

il limite superiore
o inferiore

\hookrightarrow utilizz qui
un limite superiore



non cresce
nello stesso modo

questa complessità
è lineare

Complessità computazionale

→ costo di un programma

Definizione $O()$

$$T(n) \in O(f(n))$$

$$T(n) \leq f(n)$$

cresce di meno per n grande

vuo dire che esistono due costanti:

$$c \mid \text{se } n \geq n_0$$

a partire da n_0

$T(n) \leq c f(n)$ questa condizione è vera solo quando

$$\underline{n \geq n_0}$$

$$T(n) = cn + q \in O(n)$$

dobbiamo trovare

$$T(n) \leq cn$$

sia vero
per $\underline{n \geq n_0}$

da un certo punto

se sceglie

$$c = 40$$

ma anche

$$T(u) < c u^2$$

e
vera

$$T(u) \geq c' u \quad \text{per } u \geq u_0$$

`len(t)` #lunghezza della tupla

Spiegazione Esercizio

non mi ricordo il numero
a punta ad h perchè è un parametro della funzione

$a = [4, 2, 2, 3, 2, 4, 5]$

• $a = [4, 2, 2, 3, 2, 4, 5]$

• $b = [4, 2, 3, 4, 5]$

$a = [4, 2, 3, 4, 5, 4, 5]$

def del_item(a, v):
—
—

siccome sto aggiornando gli indici di a direttamente con

• $a[i] = b[i]$,
e non con
 $a = b$, → alias

```
def del_item(a, v): # definizione della funzione

    b = []

    for x in a:
        if x != v:
            b.append(x)

    i = 0
    while i < len(b):
        a[i] = b[i]
        i += 1

    # i è la posizione del primo elemento da eliminare da a
    while i < len(a):
        del(a[-1])

L = [4, 1, 7, 6, 5, 6, 8, 2, 3, 1, 2, 7, 8]
del_item(L, 1)
print(L)

# complessità temporale O(n)
```

1° ottimizzare il tempo
2° la memoria (spazio)

- 1° complessità temporale
- 2° complessità spaziale
- usare notazioni simili alla 1°
- funzione quanto spazio occupa?

la complessità spaziale non si deve tenere conto dell'input della funzione

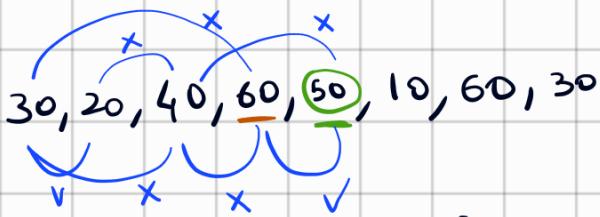
su input e output non posso farci niente e quindi dobbiamo preoccuparci dello SPAZIO SUPPLEMENTARE

$n = \ln(a)$ lista a
 $n = \ln(b)$ lista b
 $O(n)$

Trovare il secondo massimo

Input una lista

- $\max = 0$
- $\text{second_max} = 0$



ritornare
secondo
massimo

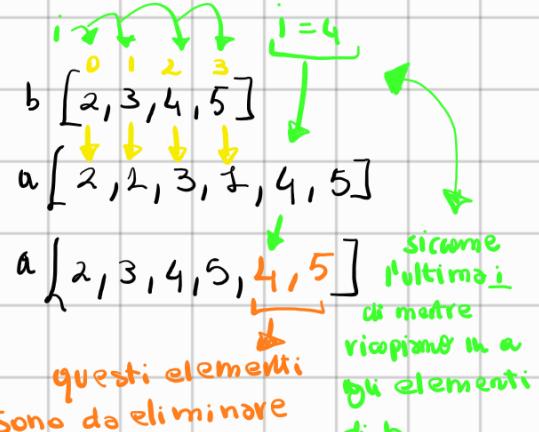
$$a = [2, 2, 3, 2, 4, 5]$$

↓ processo di copiare in b
tutti tranne v(2)

$$b = [2, 3, 4, 5]$$

↳ ricopiare nelle posizioni in a di b
ovvero arremmo che a sarà

OSS.
fino a qui abbiamo che per esempio



↳ sarà proprio
l'indice del
primo
elemento
di v
da elim.

```

    def del_item(a, v): # definizione della funzione
        b = [] # assegnazione alla variabile chiamata "b" una lista vuota
        for x in a: # inizio ciclo for (iteramento sulla lista)
            if x != v: # controllo se l'elemento x (di a) è diverso dal valore v
                b.append(x) # aggiunge allo lista b l'elemento x (solo è diverso da v)
        i = 0 # i è la posizione del primo elemento da eliminare da a
        while i < len(b): # i < len(b) è la condizione di continuazione finché i è minore o uguale alla lunghezza della lista b
            a[i] = b[i] # iniziamo da i uno dovrebbe essere il primo elem. da eliminare e usiamo del(-1)
            i += 1 # incrementatore i
        del(a[-1]) # eliminiamo l'ultimo elemento della lista a
    L = [4, 1, 7, 6, 5, 6, 8, 2, 3, 1, 2, 7, 8]
    del_item(L, 1)
    print(L)
    # complessità temporale O(n)

```

concettore
per il ciclo
while

il ciclo while
con condizione

continua finché
i è minore stretto
della lunghezza
della lista a
 $i < \text{len}(a)$

test della funzione
che abbiamo definito
ricordiamo qui siamo
FUORI (GLOBAL)
e non PI
all'interno
della funzione

↳ siccome del elimina
e diciamo che scalo
possiamo direttamente
diminuire l'ultimo
o ricordiamo che
list[-1] è l'ultimo
elemento di una
sequenza

