lezione06

October 12, 2023

1 Fondamenti di Programmazione

Andrea Sterbini

lezione 6 - 12 ottobre 2023

1.1 RECAP

- metodi dei contenitori **list, tuple, set, dict**
- 2 quesiti con la Susi
- assegnamenti multipli, packing ed unpacking

1.2 Input e stampa

Per prendere dei dati da tastiera si usa la funzione **input([prompt])** che: - prende come argomento un "prompt" opzionale (un testo che viene mostrato per guidare l'utente) - torna come risultato la stringa inserita, compresa di **'\n'** finale

NOTA torna sempre una stringa, se vi serve un tipo di dato diverso dovete convertirlo

[3]:
```
## Esempio
numero = input('Inserisci un intero tra 1 e 100')
numero = int(numero)
while numero < 0 or numero > 100:
    numero = input('Inserisci un intero tra 1 e 100')
    numero = int(numero)
numero
```

```
Inserisci un intero tra 1 e 100 340
Inserisci un intero tra 1 e 100 -12
Inserisci un intero tra 1 e 100 0
```

[3]:
```
0
```

2 Come ordinare gli elementi di un contenitore

- col metodo **sort** delle liste (modifica distruttiva)
- con la funzione **sorted(contenitore) -> list**

L'ordinamento applicato è quello **crescente** naturale per il tipo di dato (crescente numerico per gli **int** e **float** oppure crescente alfabetico per le **str**)

[7]:
```
# Esempio
L = [ 'uno', 'due', 'tre', 'quattro', 'cinque', 'sei', 'sette', 'otto', 'nove',␣
 ↪'dieci']
print(sorted(L))
L.sort()
L
```

```
['cinque', 'dieci', 'due', 'nove', 'otto', 'quattro', 'sei', 'sette', 'tre',
'uno']
```

[7]:
```
['cinque',
 'dieci',
 'due',
 'nove',
 'otto',
 'quattro',
 'sei',
 'sette',
 'tre',
 'uno']
```

NOTA tutti gli elementi da ordinare devono essere **confrontabili** - **OK** interi e float - **OK** stringhe - **NOT OK** interi e stringhe

[14]:
```
## Otteniamo l'ordine OPPOSTO con l'argomento opzionale reverse=True
print(sorted(L, reverse=True))
```

```
['uno', 'tre', 'sette', 'sette', 'sei', 'due', 'cinque']
```

2.1 Ordinamenti complessi

Come fare per ordinarle in modo più sofisticato?

Esempio: `[ 'uno', 'due', 'tre', 'quattro', 'cinque', 'sei', 'sette']` : - per **lunghezza crescente delle parole** - e se sono di lunghezza uguale, **in ordine alfabetico**

Potrei **trasformare ciascun elemento in una coppia**: - lunghezza della parola - parola

e cercare di ordinare la nuova lista di coppie

[9]:
```
L = [ 'uno', 'due', 'tre', 'quattro', 'cinque', 'sei', 'sette']
# costruisco la lista di coppie
L1 = []
for elemento in L:
    L1.append((len(elemento), elemento))
print(L1)
# la ordino
L1.sort()
print(L1)
```

```
[(3, 'uno'), (3, 'due'), (3, 'tre'), (7, 'quattro'), (6, 'cinque'), (3, 'sei'),
(5, 'sette')]
[(3, 'due'), (3, 'sei'), (3, 'tre'), (3, 'uno'), (5, 'sette'), (6, 'cinque'),
(7, 'quattro')]
```

2.1.1 FUNZIONA! MA PERCHE'?

Perchè per confrontare due coppie: - prima confrontiamo il primo elemento (lunghezza della parole) - poi solo in caso di parità passiamo a confrontare il secondo elemento (ordine alfabetico delle parole)

... e questo era proprio quello che volevamo!

[10]:
```
# a questo punto basta estrarre dalle coppie solo la parola originale
L2 = []
for lunghezza,elemento in L1:
    L2.append(elemento)
L2
```

[10]:
```
['due', 'sei', 'tre', 'uno', 'sette', 'cinque', 'quattro']
```

2.2 Ma se volessi che l'ordinamento fosse STABILE ?

Ovvero che elementi "uguali" che erano in un certo ordine relativo

mantengano lo stesso ordine relativo

BASTA aggiungere alla tupla anche la posizione originale

[215]:
```
## Esempio
L = [ 'uno', 'due', 'tre', 'sette', 'cinque', 'sei', 'sette']
# costruisco la lista di coppie
L1 = []
# assieme all'elemento ne estraggo la posizione con enumerate
for i,elemento in enumerate(L):
     # aggiungo la posizione *i* come ulteriore criterio da usare DOPO gli altri
    L1.append((len(elemento), elemento, i))
L1
# la ordino
L1.sort()
L1
```

[215]:
```
[(3, 'due', 1),
 (3, 'sei', 5),
 (3, 'tre', 2),
 (3, 'uno', 0),
 (5, 'sette', 3),
 (5, 'sette', 6),
 (6, 'cinque', 4)]
```

2.2.1 Questo tipo di trasformazione di ciascun elemento si chiama trasformazione di Schwartz

e può essere semplificata introducendo una piccola funzione che trasforma ciascun elemento da passare a **sorted** (che genera sempre un ordinamento stabile)

[219]:
```
# trasformazione di un solo elemento nella coppia corrispondente
def trasforma_elemento(parola):
    return len(parola), parola
# con la funzione *sorted* costruiamo una nuova lista ordinata
# fornendo col parametro *key* la funzione di trasformazione
sorted(L, key=trasforma_elemento)
# NOTA: non c'è bisogno di mettere la posizione, visto che *sorted* è stabile␣
 ↪(ci pensa lei)
```

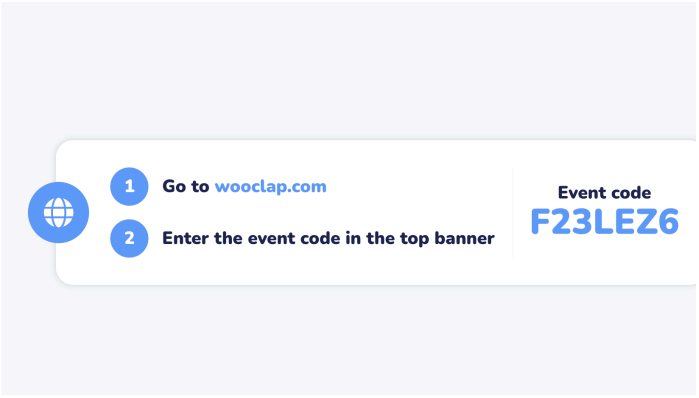
[219]:
```
['due', 'sei', 'tre', 'uno', 'sette', 'sette', 'cinque']
```

2.3 Proviamo di nuovo, ma con un ordinamento più complicato

- in ordine **crescente** di lunghezza
- in caso di parità **alfabetico ignorando il case**
- in caso di parità **alfebetico**

[220]:
```
def criterio_di_ordinamento(elemento):
    return ( len(elemento),        # per prima cosa la lunghezza deve crescere
             elemento.lower(),     # poi in ordine alfabetico CASELESS
             elemento )            # poi in ordine alfabetico
L = [ 'PaPerino', 'plUTO', 'TopoLINO', 'minniE', 'PLUTO', 'papeRINO',␣
 ↪'papEROne', 'gasTONE', 'MInnie']
S = sorted(L, key=criterio_di_ordinamento, reverse=True)
```

2.4 Wooclap: secondo voi cosa stampa?



[22]:
```
print(S)
```

```
['TopoLINO', 'papEROne', 'papeRINO', 'PaPerino', 'gasTONE', 'minniE', 'MInnie',
'plUTO', 'PLUTO']
```

[ ]:
```
# Soluzione
A = [ '']
```

MA INVECE che definire una nuova funzione apposta (che magari ci serve solo in questa occasione)

potrebbe essere comodo avere un modo per creare funzioni **USA e GETTA**

2.5 Funzioni anonime (espressioni *lambda*)

Notate che la funzione precedente è estremamente semplice ed ha un solo compito: - riceve dei parametri - ritorna SOLO una espressione **SENZA ESEGUIRE ALTRE ISTRUZIONI**

Quando calcolate **una sola espressione** potete scriverla senza usare la keyword **return** come

```
lambda argomenti: espressione
```

NOTA l'espressione può produrre una lista o una tupla di valori, che è proprio quello che ci serve per **sorted**

[34]:
```
# quindi la funzione *criterio_di_ordinamento*
# può essere scritta direttamente come
#
criterio_di_ordinamento = lambda elemento: (len(elemento),elemento.lower(),␣
 ↪elemento)

print(criterio_di_ordinamento('PaPErino'))

#
S = sorted(L, key=lambda elemento: (len(elemento),elemento.lower(), elemento),␣
 ↪reverse=True)
print(S)
```

```
(8, 'paperino', 'PaPErino')
['TopoLINO', 'papEROne', 'papeRINO', 'PaPerino', 'gasTONE', 'minniE', 'MInnie',
'plUTO', 'PLUTO']
```

2.5.1 RIASSUNTO

Per ordinare un contenitore di elementi secondo criteri complessi: - si definisce una funzione di trasformazione (o una lambda) che: - riceve l'elemento da trasformare (**UN SOLO ARGOMENTO**) - torna una tupla contenente nell'ordine i dati che servono a rappresentare i criteri complessi

Esempio: ordiniamo la lista di parole - in ordine **CRESCENTE** di lunghezza - e in caso di parità in ordine alfabetico **DECRESCENTE** (Z->A)

qui abbiamo un problema in più, **i due ordinamenti vanno in direzioni opposte!**

Uno dei due va rovesciato, perchè i confronti tra tuple sono sempre monodirezionali cioè tutti gli elementi delle tuple vengono confrontati nella stessa direzione

NON POSSO rovesciare l'ordinamento alfabetico

POSSO rovesciare l'ordinamento dei numeri **CAMBIANDOGLI SEGNO**!

[38]:
```
# definiamo la funzione di trasformazione adatta
# NON possiamo rovesciare l'ordinamento alfabetico!
# ALLORA rovesciamo l'ordinamento delle lunghezze
def trasforma_elemento(el):
    return -len(el), el          # lunghezza negativa: valori più grandi␣
 ↪saranno più a sinistra
```

[222]:
```
print('disordinata', L)
# vediamo cosa succede
S = sorted(L, key=trasforma_elemento)
# E' tutto a rovescio!
print('ordinata', S)
```

```
disordinata ['PaPerino', 'plUTO', 'TopoLINO', 'minniE', 'PLUTO', 'papeRINO',
'papEROne', 'gasTONE', 'MInnie']
ordinata ['PLUTO', 'plUTO', 'MInnie', 'minniE', 'gasTONE', 'PaPerino',
'TopoLINO', 'papEROne', 'papeRINO']
```

[40]:
```
# cambiamo il parametro *reverse* per ordinare in direzione opposta
S = sorted(L, key=trasforma_elemento, reverse=True)
# ora va bene
print(S)
```

```
['plUTO', 'PLUTO', 'minniE', 'MInnie', 'gasTONE', 'papeRINO', 'papEROne',
'TopoLINO', 'PaPerino']
```

[41]:
```
# A questo punto possiamo riscrivere la trasformazione
# usando una funzione anonima con lambda
# (notate le parentesi per creare la tupla)
S = sorted(L, key=lambda el: (-len(el),el), reverse=True)
print(S)
```

```
['plUTO', 'PLUTO', 'minniE', 'MInnie', 'gasTONE', 'papeRINO', 'papEROne',
'TopoLINO', 'PaPerino']
```

[59]:
```
# Soluzione

# Supponiamo di sapere età
# e genere di alcuni personaggi
L = [ (27, 'M', 'Paperino'),
      (31, 'M', 'Topolino'),
      (26, 'F', 'Paperina'),
      (32, 'F', 'Minnie')]
def trasforma_elemento(terna):
    eta, genere, nome = terna
    return len(nome), genere, eta
sorted(L, key=trasforma_elemento)
```

[59]:
```
[(32, 'F', 'Minnie'),
 (26, 'F', 'Paperina'),
 (27, 'M', 'Paperino'),
 (31, 'M', 'Topolino')]
```

3 List comprehension

3.1 una sintassi semplificata per costruire contenitori

Spessissimo dobbiamo raccogliere in un contenitore più elementi e scriviamo roba del tipo di

- iniziamo costruendo una lista vuota
- iteriamo su un contenitore di elementi
- trasformiamo ciascun elemento
- lo aggiungiamo alla lista
- torniamo la lista ottenuta

Sarebbe bello poter usare una sintassi più leggibile

(che non ci obbliga a **simulare** il codice nella testa, simile alle espressioni insiemistiche)

[42]:
```
# dati dei valori
lista_di_interi = [12, 34, 61, 2, 4, 7]
# voglio costruire la lista dei cubi
lista_di_cubi = []                  # inizio con una lista vuota
for x in lista_di_interi:           # scandisco i valori
    lista_di_cubi.append(x**3)      # aggiungo in fondo il cubo del valore␣
 ↪corrente
lista_di_cubi
```

[42]:
```
[1728, 39304, 226981, 8, 64, 343]
```

3.1.1 List-comprehension: sintassi

```
[ espressione                       # valore calcolato per il risultato
    for variabile in contenitore    # per ciascun valore di contenitore di dati
]
```

[44]:
```
lista_di_interi = [12, 34, 2, 61, 2, 4, 7]

# Con la sintassi della list-comprehension
lista_di_cubi = [ X**3 for X in lista_di_interi ]

# - le parentesi indicano che tipo di contenitore stiamo costruendo (lista)
# - il *for* indica su quale sequenza di dati stiamo iterando
# - l'espressione all'inizio (X**3) indica come produciamo ogni nuovo valore
lista_di_cubi
```

[44]:
```
[1728, 39304, 8, 226981, 8, 64, 343]
```

[20]:
```
# possiamo  costruire altri tipi di contenitore
# INSIEMI, racchiusi tra parentesi graffe
{ X**3 for X in lista_di_interi }
```

[20]:
```
{8, 64, 343, 1728, 39304, 226981}
```

[223]:
```
# DIZIONARI che contengono coppie chiave:valore racchiuse tra parentesi graffe
{ X : X**3 for X in lista_di_interi }
```

[223]:
```
{12: 1728, 34: 39304, 2: 8, 61: 226981, 4: 64, 7: 343}
```

[46]:
```
# TUPLE (qui ci vuole la parola tuple, non bastano le parentesi)
tuple( X**3 for X in lista_di_interi )
```

[46]:
```
(1728, 39304, 8, 226981, 8, 64, 343)
```

3.1.2 e possiamo anche condizionare quali elementi trasformare e quali no

```
[ espressione                       # valore calcolato
     for variabile in contenitore   # per ciascun valore di contenitore
     if condizione                  # ma solo se la condizione è vera
]
```

[224]:
```
# Voglio i cubi MA SOLO dei numeri dispari
{ x : x**3 for x in lista_di_interi if x%2!=0 }
```

[224]:
```
{61: 226981, 7: 343}
```

3.1.3 oppure costruire for nidificati

```
[ espressione                   # valore calcolato
     for var1 in contenitore1   # per ciascun valore di contenitore1
     for var2 in contenitore2   # per ciascun valore di contenitore2
]
```

[50]:
```
# Esempio: costruisco l'insieme dei prodotti della tabellina del 10
S = { X*Y  for X in range(1,11)
           for Y in range(1,11) }
print(S)
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 24, 25, 27, 28, 30,
32, 35, 36, 40, 42, 45, 48, 49, 50, 54, 56, 60, 63, 64, 70, 72, 80, 81, 90, 100}
```

3.1.4 oppure costruire contenitori nidificati

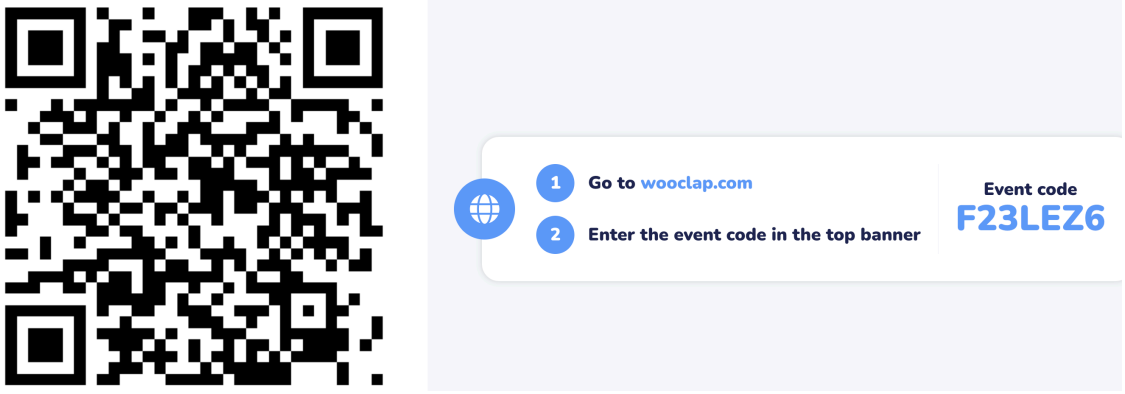
```
[ [ espressione for var1 in contenitore1 ]   # costruisco una lista
                for var2 in contenitore2 ]   # per ciascun valore del contenitore2
```

[58]:
```
# Ad esempio la tabellina del 10 come lista di liste
# (una lista per ogni riga)
T = [ [ X*Y for X in range(1,11) ]  # per ogni Y costruisco la riga con 10␣
 ↪moltiplicazioni
            for Y in range(1,11) ]
%pprint
T
```

```
Pretty printing has been turned ON
```

[58]:
```
[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
 [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
 [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
 [4, 8, 12, 16, 20, 24, 28, 32, 36, 40],
 [5, 10, 15, 20, 25, 30, 35, 40, 45, 50],
 [6, 12, 18, 24, 30, 36, 42, 48, 54, 60],
 [7, 14, 21, 28, 35, 42, 49, 56, 63, 70],
 [8, 16, 24, 32, 40, 48, 56, 64, 72, 80],
 [9, 18, 27, 36, 45, 54, 63, 72, 81, 90],
 [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]]
```

4 Wooclap: Secondo voi cosa ottengo da questa list comprehension?



[64]:
```
## Vediamo la soluzione

alfabeto = "abcdefghijk"
parola   = "bigia"

[ alfabeto[alfabeto.index(X):]   # il pezzo di alfabeto dalla X in poi
      for X in parola]           # per ciascuna lettera X di parola
```

[64]:
```
['bcdefghijk', 'ijk', 'ghijk', 'ijk', 'abcdefghijk']
```

5 INTERVALLO (RAI - anni 70)

[225]:
```
%%html
<iframe src="https://www.youtube.com/embed/Y344g_3KSRs" allowfullscreen />
```

```
<IPython.core.display.HTML object>
```

6 COME analizzare un problema?

6.1 Strategia di analisi top-down

- cerchiamo di capire come lo faremmo su carta
  - descrivendo gli **input ed output**
  - eventuali **controlli da fare sui dati**
  - possibili **errori da generare**
  - possibili **effetti collaterali**
  - possibili **scelte non indicate**
- lo dividiamo in **problemi più piccoli**
  - ripetendo il metodo di analisi
- **continuando a frammentarlo** finchè
  - la sua **descrizione è così semplice**
  - da poter essere **implementato**
- mano a mano **testiamo le sottofunzioni** realizzate
  - **semplificando enormemente il debug** che diventa più **veloce**

7 Esempio di problema: trovare i k massimi di una lista L di valori numerici

- **rappresentazione dei dati**? (in questo caso lo sappiamo)
  - INPUT: una lista di interi ed un valore intero K
  - OUTPUT: la lista dei K massimi valori
- controlliamo se ci sono **condizioni di validità** dei dati
  - cosa fare se **K negativo**? ERRORE?
  - cosa fare se **K > len(L)**?
    - diamo **errore**?
    - torniamo un numero di elementi **minori di K**?
- ci sono **effetti collaterali**?
  - **modifichiamo distruttivamente** L ?
  - non la modifichiamo?
- vogliamo **caratteristiche particolari** del risultato?
  - **ordinato**?
  - **disordinato**?

7.1 Scegliamo ad esempio di modificare L distruttivamente

- se K è sbagliato diamo un **errore**

- **modifichiamo distruttivamente** L
- il risultato può essere in **qualsiasi ordine**

- Per estrarre i **K** massimi dalla lista **L**
  - controllo che gli argomenti siano validi
  - ripeto **K** volte
    - trovo il massimo di **L** e lo tolgo
    - lo aggiungo al risultato
  - ritorno il risultato

NOTA la modifica distruttiva toglie di torno il massimo corrente e facilita il ritrovamento del successivo

[71]:
```
# per estrarre i k massimi da L
def k_massimi_distruttivo(L, k):
    # controllo che k sia valido e altrimenti do errore con un messaggio␣
 ↪esplicativo
    assert len(L)>0,      "L è vuota"
    assert 0<k<=len(L), f"K={k} non è compreso tra 1 e len(L)={len(L)}"
    # definisco la variabile risultato e la inizializzo = []
    risultato = []
    # ripeto per k volte (l'indice non mi interessa)
    for _ in range(k):
        # estraggo un elemento massimo da L eliminandolo dalla lista
        M = estrai_massimo(L)
        # aggiungo questo elemento al risultato
        risultato.append(M)
    # torno il risultato
    return risultato
```

- per estrarre ed eliminare il massimo
  - lo cerco (con **max**)
  - lo elimino (con **remove**)
  - lo torno come risultato

[68]:
```
# per estrarre ed eliminare un massimo da una lista
def estrai_massimo(L):
    M = max(L)
    L.remove(M)
    return M

# Esempio
X = [1, 5, 2, 89, 2, 23]
estrai_massimo(X), X
```

[68]:
```
(89, [1, 5, 2, 2, 23])
```

[69]:
```
# costruiamo una lista di valori casuali (con ripetizioni)
from random import choices
```

```
help(choices)
L = list(choices(range(1000000), k=10))
print(L)
```

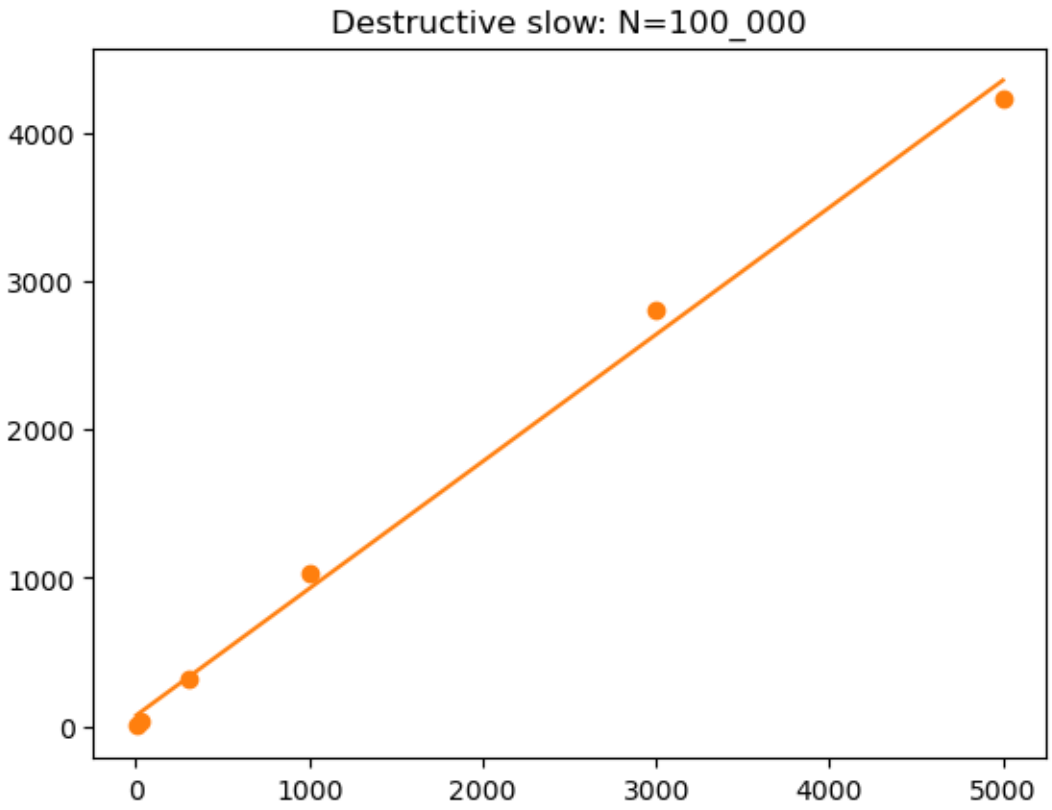
```
Help on method choices in module random:

choices(population, weights=None, *, cum_weights=None, k=1) method of
random.Random instance
    Return a k sized list of population elements chosen with replacement.

    If the relative weights or cumulative weights are not specified,
    the selections are made with equal probability.

[512330, 240375, 913996, 54098, 603240, 4500, 953490, 569646, 117877, 212029]
```

[150]:
```
## vediamo quanto tempo impiega per diversi valori di K e per N=100_000
LL = list(choices(range(1_000_000), k=100_000))
L = LL.copy()
%timeit k_massimi_distruttivo(L, 3)
L = LL.copy()
%timeit k_massimi_distruttivo(L, 30)
L = LL.copy()
%timeit k_massimi_distruttivo(L, 300)
L = LL.copy()
%timeit k_massimi_distruttivo(L, 1000)
L = LL.copy()
%timeit k_massimi_distruttivo(L, 3000)
L = LL.copy()
%timeit k_massimi_distruttivo(L, 5000)
None
```

```
3.16 ms ± 23.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
31.9 ms ± 368 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
318 ms ± 4.38 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
1.03 s ± 21.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
2.81 s ± 189 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
4.23 s ± 536 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[162]:
```
%matplotlib inline
import matplotlib.pyplot as plt
from scipy import stats
xdata = [3, 30, 300, 1000, 3000, 5000]
ydata_slow = [3.16, 31.9, 318, 1030, 2810, 4230 ]
slope, intercept, *_ = stats.linregress(xdata, ydata_slow)
plt.figure()
plt.title("Destructive slow: N=100_000")
plt.scatter(xdata, ydata_slow, color='tab:orange')
plt.plot(xdata, [x*slope+intercept for x in xdata], color='tab:orange')
```

```
None
```



Come vedete il tempo di esecuzione è proporzionale a **K**.

Per essere più precisi, se **N = len(L)**, nel caso peggiore: - per **K** volte - trovo il massimo (scandendo la lista, tempo proporzionale a **N**) - lo elimino (scandendo la lista, tempo proporzionale a **N**) - lo aggiungo al risultato (tempo costante)

Quindi il tempo nel caso peggiore è proporzionale a **K * N**

7.2 E se invece volessimo lasciare L invariata?

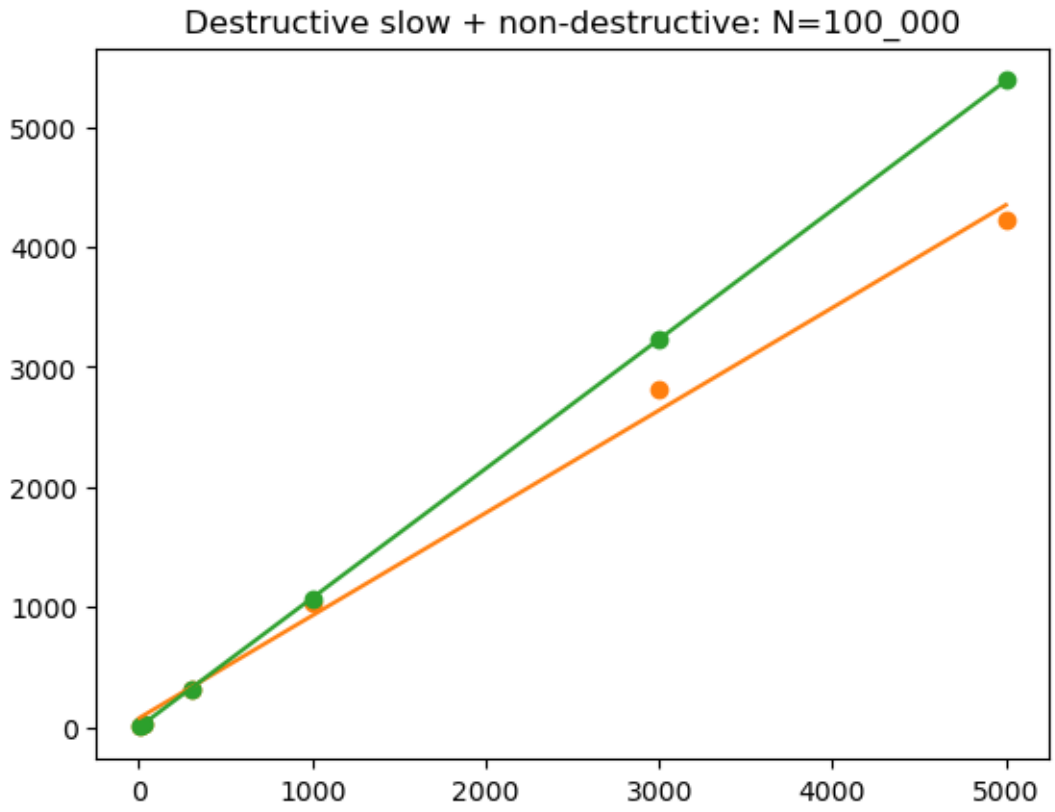
Basta copiarla

[159]:
```
# per calcolare i k_massimi SENZA modificare L
def k_massimi(L, k):
    # copio la lista                tempo proporzionale a N
    L1 = L.copy()
    # uso k_massimi sulla copia     tempo proporzionale a k*N
    return k_massimi_distruttivo(L1, k)
# Il tempo di questa implementazione è proporzionale a k*N nel caso peggiore
```

[158]:
```
# di nuovo vediamo i tempi al variare di K e/o N
L = LL.copy()
%timeit k_massimi(L, 3)
%timeit k_massimi(L, 30)
%timeit k_massimi(L, 300)
%timeit k_massimi(L, 1000)
%timeit k_massimi(L, 3000)
%timeit k_massimi(L, 5000)
None
```

```
3.64 ms ± 18 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
32.5 ms ± 93.7 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
321 ms ± 1.36 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
1.07 s ± 3.76 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
3.23 s ± 15.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
5.39 s ± 40.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[214]:
```
xdata = [3, 30, 300, 1000, 3000, 5000]
ydata_slow2 = [3.64, 32.5, 321, 1070, 3230, 5390 ]
slope2, intercept2, *_ = stats.linregress(xdata, ydata_slow2)
plt.figure()
plt.title("Destructive slow + non-destructive: N=100_000")
plt.scatter(xdata, ydata_slow, color='tab:orange')
plt.plot(xdata, [x*slope+intercept for x in xdata], color='tab:orange')
plt.scatter(xdata, ydata_slow2, color='tab:green')
plt.plot(xdata, [x*slope2+intercept2 for x in xdata], color='tab:green')
None
```



7.3 Seconda idea: ordiniamo L

Per estrarre i primi k massimi da L (**NOTA** non è distruttivo) - controllo se k è valido - costruisco la lista ordinata di tutti i valori in ordine decrescente - estraggo e torno i primi k

[99]:
```
# Per estrarre i primi k massimi da L (non distruttivo)
def k_massimi_sorted(L, k):
    # controllo se k è valido        tempo costante
    assert 0<k<=len(L)
    # ordino tutti i valori in ordine decrescente    tempo O(N*log(N))
    ordinata = sorted(L,reverse=True)
    # estraggo e torno i primi k     tempo O(k)
    return ordinata[:k]
```

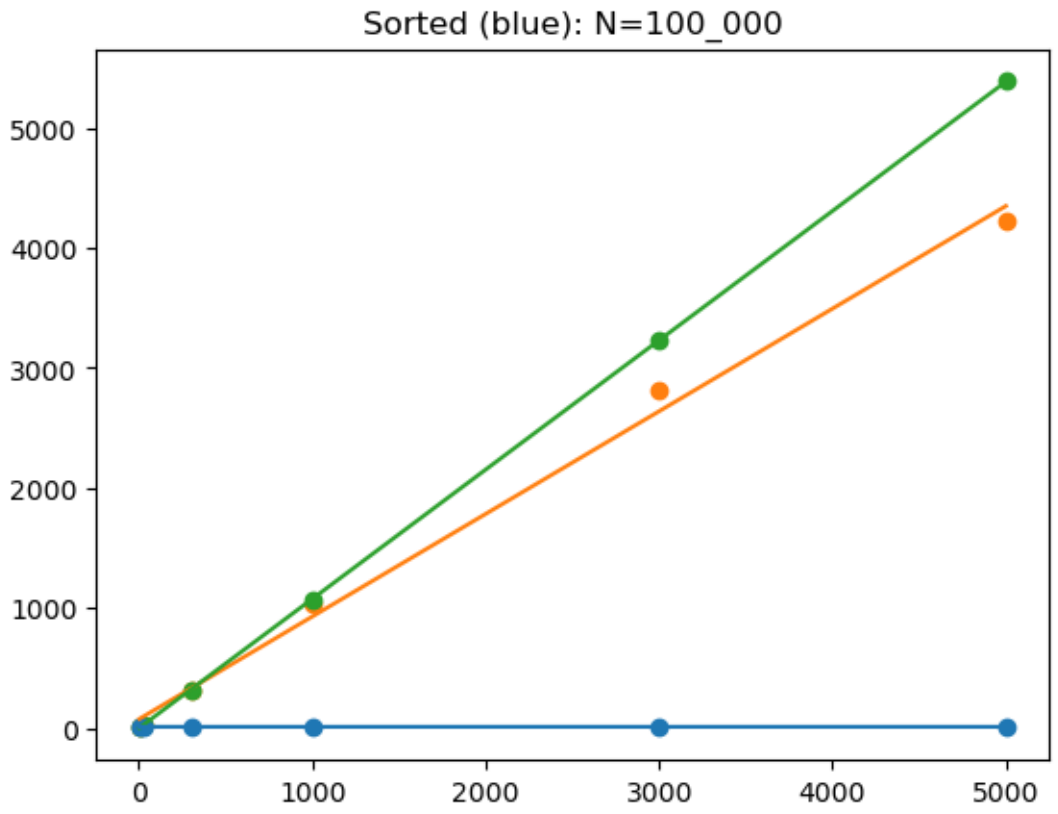
[166]:
```
%timeit k_massimi_sorted(L, 3)
%timeit k_massimi_sorted(L, 30)
%timeit k_massimi_sorted(L, 300)
%timeit k_massimi_sorted(L, 1000)
%timeit k_massimi_sorted(L, 3000)
%timeit k_massimi_sorted(L, 5000)
None
```

```
9.33 ms ± 183 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
9.55 ms ± 562 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
9.22 ms ± 21.6 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
9.22 ms ± 17.9 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
9.23 ms ± 18.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
9.31 ms ± 31.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Vedete che stavolta il tempo quasi non è dipeso da **K**, infatti - creare la lista ordinata impiega tempo proporzionale a $N * log2(N)$ - estrarre i **K** massimi impiega tempo proporzionale a **K**

Quindi il tempo totale è proporzionale a $K + N * log(N)$

[168]:
```
xdata = [3, 30, 300, 1000, 3000, 5000]
ydata_sorted = [9.33, 9.55, 9.22, 9.22, 9.23, 9.31 ]
slope3, intercept3, *_ = stats.linregress(xdata, ydata_sorted)
plt.figure()
plt.title("Sorted (blue): N=100_000")
plt.scatter(xdata, ydata_slow, color='tab:orange')
plt.plot(xdata, [x*slope+intercept for x in xdata], color='tab:orange')
plt.scatter(xdata, ydata_slow2, color='tab:green')
plt.plot(xdata, [x*slope2+intercept2 for x in xdata], color='tab:green')
plt.scatter(xdata, ydata_sorted, color='tab:blue')
plt.plot(xdata, [x*slope3+intercept3 for x in xdata], color='tab:blue')
None
```



7.3.1 Ma quanta memoria stiamo usando?

- la lista **L** occupa uno spazio proporzionale a **N**
- la lista ordinata occupa uno spazio proporzionale ad **N**

Ce n'è proprio bisogno? What if dobbiamo trovare i **K** massimi in uno stream di milioni di dati? (un sensore, per esempio)

In realtà è sufficiente "ricordare" solo gli ultimi **K** valori massimi

Per estrarre i **K** massimi da uno stream di **N** dati - all'inizio i massimi sono una lista vuota - per ogni dato **X** letto - aggiorno l'elenco degli ultimi **K** massimi incontrati

[125]:
```
def k_massimi_lowmem(K, L):
    massimi = []
    for X in L:                          # per N volte
        update_massimi(massimi, K, X)    # tempo ?
    return massimi

# ci aspettiamo un tempo proporzionale a N volte il tempo di update_massimi
```

Per aggiornare gli ultimi **K** massimi con un nuovo valore **X** - se ho meno di **K** valori - aggiungo **X** ai massimi - altrimenti sono K, e se **X** è minore o uguale al più piccolo dei massimi - lo posso tranquillamente ignorare - altrimenti - tolgo il minimo dai massimi - aggiungo **X** ai massimi

[173]:
```
def update_massimi(massimi_correnti, k, X):
    if k > len(massimi_correnti):            # tempo costante
        massimi_correnti.append(X)           # tempo costante
        return
    minimo = min(massimi_correnti)           # proporzionale a K
    if X <= minimo:
        return
    else:
        massimi_correnti.remove(minimo)      # proporzionale a K
        massimi_correnti.append(X)           # tempo costante

# ci aspettiamo nel caso peggiore tempo proporzionale a K
```

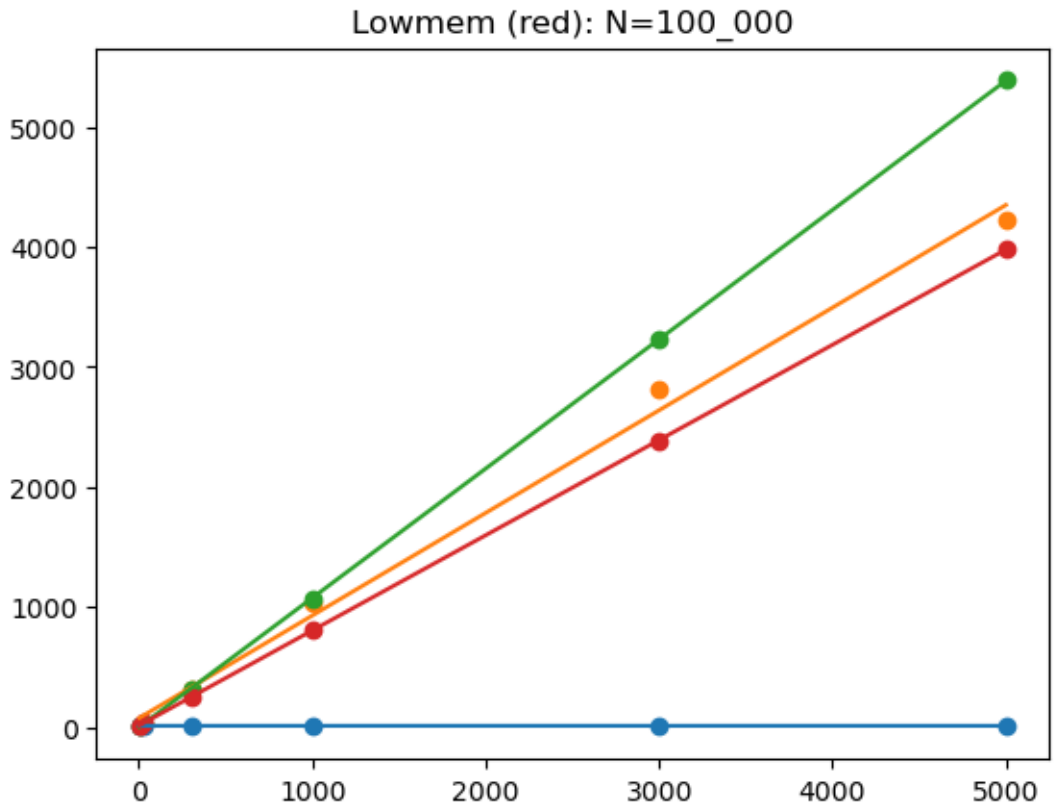
[174]:
```
# Esempio di aggiornamento
massimi = [12, 45, 67, 1, 90, -2, 17]
print('massimi iniziali\t',massimi)
update_massimi(massimi, 10, 51)
print('meno di K=10 massimi\t', massimi)
update_massimi(massimi, 8, -29)
print('X minore del minimo\t',massimi)
update_massimi(massimi, 8, 23)
print('X maggiore del minimo\t',massimi)
```

```
massimi iniziali          [12, 45, 67, 1, 90, -2, 17]
meno di K=10 massimi      [12, 45, 67, 1, 90, -2, 17, 51]
X minore del minimo       [12, 45, 67, 1, 90, -2, 17, 51]
X maggiore del minimo     [12, 45, 67, 1, 90, 17, 51, 23]
```

[175]:
```
L = LL.copy()
%timeit k_massimi_lowmem(3,    L)
%timeit k_massimi_lowmem(30,   L)
%timeit k_massimi_lowmem(300,  L)
%timeit k_massimi_lowmem(1000, L)
%timeit k_massimi_lowmem(3000, L)
%timeit k_massimi_lowmem(5000, L)
```

```
13.5 ms ± 82.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
35.3 ms ± 133 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
244 ms ± 903 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
808 ms ± 4.28 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
2.39 s ± 4.15 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
3.98 s ± 3.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[176]:
```
xdata = [3, 30, 300, 1000, 3000, 5000]
ydata_lowmem1 = [13.5, 35.3, 244, 808, 2390, 3980 ]
slope4, intercept4, *_ = stats.linregress(xdata, ydata_lowmem1)
plt.figure()
plt.title("Lowmem (red): N=100_000")
plt.scatter(xdata, ydata_slow, color='tab:orange')
plt.plot(xdata, [x*slope+intercept for x in xdata], color='tab:orange')
plt.scatter(xdata, ydata_slow2, color='tab:green')
plt.plot(xdata, [x*slope2+intercept2 for x in xdata], color='tab:green')
plt.scatter(xdata, ydata_sorted, color='tab:blue')
plt.plot(xdata, [x*slope3+intercept3 for x in xdata], color='tab:blue')
plt.scatter(xdata, ydata_lowmem1, color='tab:red')
plt.plot(xdata, [x*slope4+intercept4 for x in xdata], color='tab:red')
None
```



7.4 miglioriamo la gestione dei K massimi

che succede se li manteniamo ordinati?

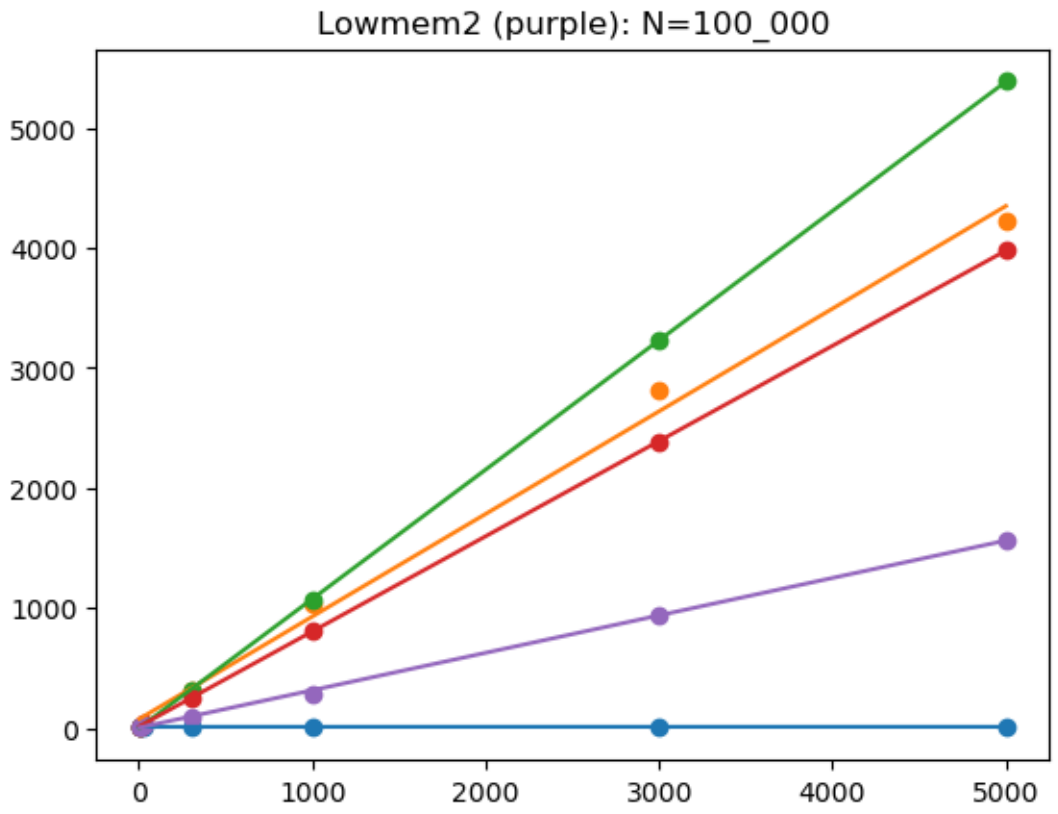
Per aggiornare i **K** massimi - trovare il minimo diventa costante (è sempre in fondo) - eliminare il minimo lo stesso - mantenere i **K** valori ordinati impiega? - $K * log(K)$ se uso **sort** - $K + log(K)$ se cerco la posizione con una ricerca binaria ($log(K)$) e poi inserisco il valore (tempo **K**) (**MEGLIO**)

[177]:
```
## Proviamo quello più semplice da scrivere
def update_massimi(massimi, K, X):
    massimi.append(X)                # tempo costante
    massimi.sort(reverse=True)       # tempo K*log(K)
    massimi[K:] = []                 # elimino i valori oltre la posizione K se␣
 ↪esistono, tempo costante (al massimo è 1)
```

[178]:
```
%timeit k_massimi_lowmem(3,    L)
%timeit k_massimi_lowmem(30,   L)
%timeit k_massimi_lowmem(300,  L)
%timeit k_massimi_lowmem(1000, L)
%timeit k_massimi_lowmem(3000, L)
%timeit k_massimi_lowmem(5000, L)
```

```
15.4 ms ± 124 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
23 ms ± 59.3 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
95.6 ms ± 509 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
287 ms ± 2.12 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
934 ms ± 4.05 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
1.57 s ± 7.39 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[183]:
```
xdata = [3, 30, 300, 1000, 3000, 5000]
ydata_lowmem2 = [15.4, 23, 95.6, 287, 934, 1570 ]
slope5, intercept5, *_ = stats.linregress(xdata, ydata_lowmem2)
plt.figure()
plt.title("Lowmem2 (purple): N=100_000")
plt.scatter(xdata, ydata_slow, color='tab:orange')
plt.plot(xdata, [x*slope+intercept for x in xdata], color='tab:orange')
plt.scatter(xdata, ydata_slow2, color='tab:green')
plt.plot(xdata, [x*slope2+intercept2 for x in xdata], color='tab:green')
plt.scatter(xdata, ydata_sorted, color='tab:blue')
plt.plot(xdata, [x*slope3+intercept3 for x in xdata], color='tab:blue')
plt.scatter(xdata, ydata_lowmem1, color='tab:red')
plt.plot(xdata, [x*slope4+intercept4 for x in xdata], color='tab:red')
plt.scatter(xdata, ydata_lowmem2, color='tab:purple')
plt.plot(xdata, [x*slope5+intercept5 for x in xdata], color='tab:purple')
None
```



7.5 E finalmente usando la ricerca binaria

Per aggiornare i massimi con un nuovo X mantenendo i massimi ordinati - se i massimi sono meno di K - cerco la posizione di inserimento e inserisco X - se X è minore del minimo - lo ignoro - altrimenti - elimino il minimo - cerco la posizione di inserimento e inserisco X

[186]:
```
def update_massimi(massimi, K, X):
    if len(massimi) < K:                    # se ho meno di K massimi
        pos = ricerca_binaria(massimi, X)   # trovo dove            (tempo␣
↪log(K))
        massimi.insert(pos, X)              # inserire X           (tempo␣
↪proporzionale a K)
        return
    elif X >  massimi[-1]:                  # altrimenti se X > del minimo  ␣
↪(tempo costante)
        del massimi[-1]                     # elimino il minimo    (tempo␣
↪costante)
        pos = ricerca_binaria(massimi, X)   # e trovo dove         (tempo␣
↪log(K))
        massimi.insert(pos, X)              # inserire X           (tempo␣
↪proporzionale a K)
```

7.5.1 Ricerca binaria

Per cercare dove inserire un elemento X in una lista **ORDINATA** - inizializzo **inizio** e **fine** agli indici del primo e ultimo elemento - ripeto se la parte in cui cerco non è vuota (ovvero se inizio <= fine) - se il valore centrale == X - la posizione giusta è l'indice del centrale, la ritorno - se è minore di X - devo cercare a DESTRA (voglio l'ordinamento decrescente) - aggiorno inizio = centrale+1 - altrimenti - devo cercare a SINISTRA - aggiorno fine = centrale-1 Se non ho trovato il valore la posizione in cui va inserito è quella dell'inizio

[188]:
```
def ricerca_binaria(Lista, Valore):
    inizio = 0
    fine = len(Lista)-1
    while inizio <= fine:
        centrale = (inizio + fine)//2
        Y = Lista[centrale]
        if Valore == Y:
            return centrale
        elif Valore < Y:
            inizio = centrale+1
        else:
            fine = centrale-1
    return inizio
```

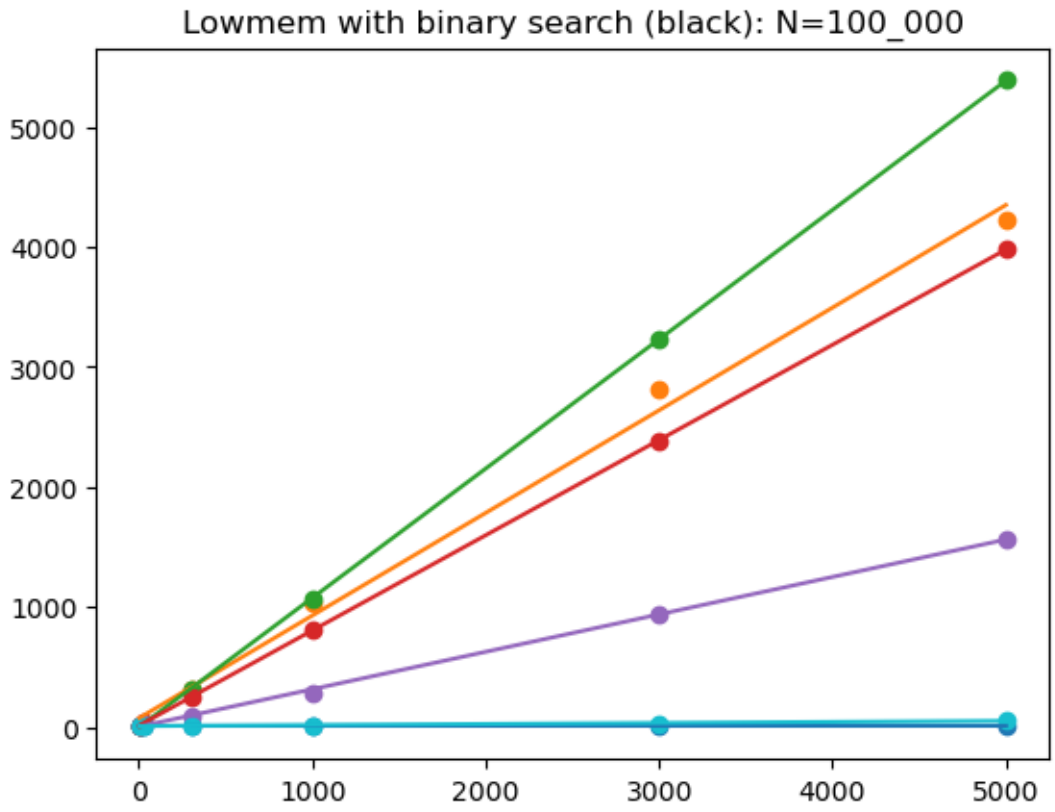
[195]:
```
# Esempio
ordinati = list(range(100,-1,-2))
ricerca_binaria(ordinati, 33)
ordinati.insert(34,33)
print(ordinati)
```

```
[100, 98, 96, 94, 92, 90, 88, 86, 84, 82, 80, 78, 76, 74, 72, 70, 68, 66, 64,
62, 60, 58, 56, 54, 52, 50, 48, 46, 44, 42, 40, 38, 36, 34, 33, 32, 30, 28, 26,
24, 22, 20, 18, 16, 14, 12, 10, 8, 6, 4, 2, 0]
```

[196]:
```
# E finalmente vediamo i tempi
%timeit k_massimi_lowmem(3,    L)
%timeit k_massimi_lowmem(30,   L)
%timeit k_massimi_lowmem(300,  L)
%timeit k_massimi_lowmem(1000, L)
%timeit k_massimi_lowmem(3000, L)
%timeit k_massimi_lowmem(5000, L)
```

```
7.83 ms ± 21.6 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
7.94 ms ± 18.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
9.98 ms ± 37.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
14.8 ms ± 80.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
31.6 ms ± 387 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
51.3 ms ± 1.02 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[198]:
```
xdata = [3, 30, 300, 1000, 3000, 5000]
ydata_lowmem3 = [7.83, 7.94, 9.98, 14.8, 31.6, 51.3 ]
slope6, intercept6, *_ = stats.linregress(xdata, ydata_lowmem3)
plt.figure()
plt.title("Lowmem with binary search (black): N=100_000")
plt.scatter(xdata, ydata_slow, color='tab:orange')
plt.plot(xdata, [x*slope+intercept for x in xdata], color='tab:orange')
plt.scatter(xdata, ydata_slow2, color='tab:green')
plt.plot(xdata, [x*slope2+intercept2 for x in xdata], color='tab:green')
plt.scatter(xdata, ydata_sorted, color='tab:blue')
plt.plot(xdata, [x*slope3+intercept3 for x in xdata], color='tab:blue')
plt.scatter(xdata, ydata_lowmem1, color='tab:red')
plt.plot(xdata, [x*slope4+intercept4 for x in xdata], color='tab:red')
plt.scatter(xdata, ydata_lowmem2, color='tab:purple')
plt.plot(xdata, [x*slope5+intercept5 for x in xdata], color='tab:purple')
plt.scatter(xdata, ydata_lowmem3, color='tab:cyan')
plt.plot(xdata, [x*slope6+intercept6 for x in xdata], color='tab:cyan')
None
```



7.5.2 Confrontiamo questa ultima con l'ordinamento di tutti i valori

- se ordiniamo tutti i valori e ne prendiamo i **K** massimi impieghiamo tempo $N * log(N)$ | **N** | $log(N)$ | | —— | ———— | | 10 | circa 3 | | 100 | circa 7 | | 1000 | circa 10 | | 1_000_000 | circa 20 |
- se manteniamo ordinati solo i **K** massimi impieghiamo nel caso peggiore tempo $N * (K + log(K)) = K * N$ (si ignora log(K))

Il secondo è **MEGLIO** se $K << log(N)$ (**E INOLTRE** usa pochissima memoria!)

Quindi l'implementazione da usare **dipende dal tipo di applicazione**.

[205]:
```
L1000 = list(choices(range(1000000),    k=1000))
L5000 = list(choices(range(1000000),    k=5000))
L10_000 = list(choices(range(1000000),  k=10_000))
L50_000 = list(choices(range(1000000),  k=50_000))
L100_000 = list(choices(range(1000000), k=100_000))
L500_000 = list(choices(range(1000000),  k=500_000))
L1_000_000 = list(choices(range(1000000), k=1_000_000))
%timeit k_massimi_sorted(L1000, 10)
%timeit k_massimi_sorted(L5000, 10)
%timeit k_massimi_sorted(L10_000, 10)
```

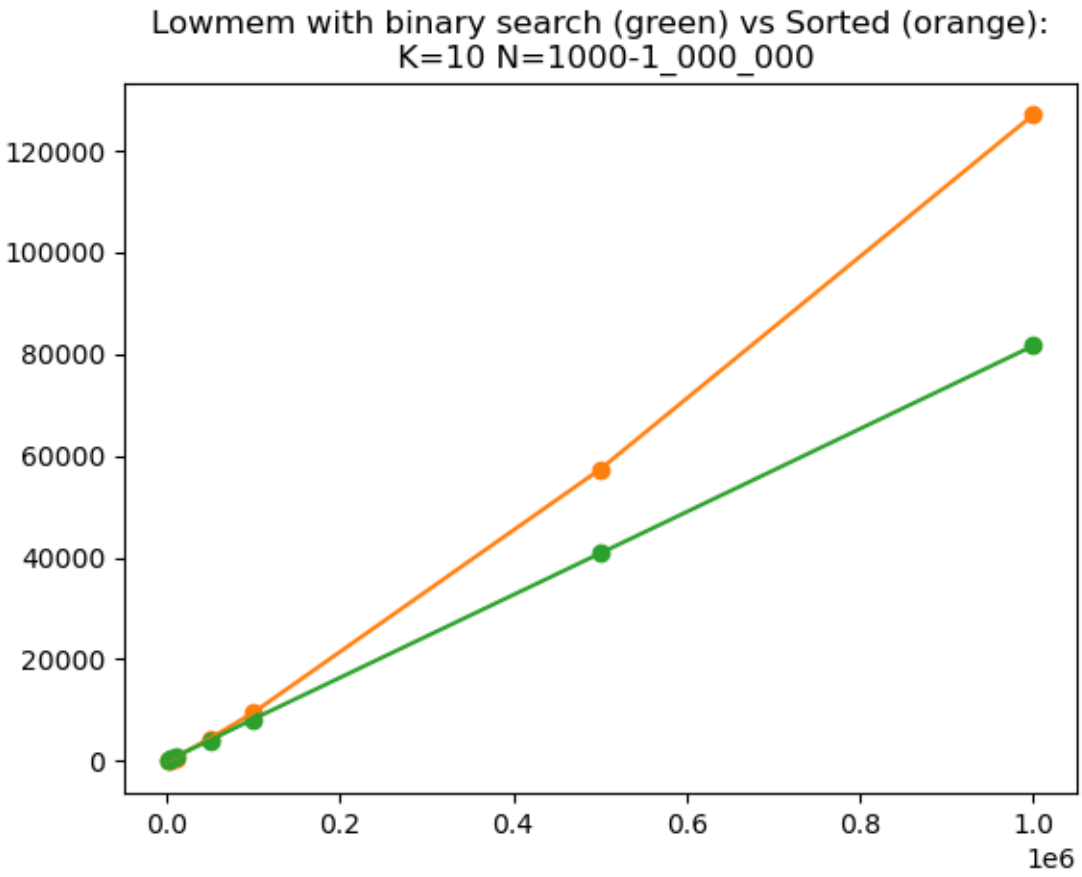
```
%timeit k_massimi_sorted(L50_000, 10)
%timeit k_massimi_sorted(L100_000, 10)
%timeit k_massimi_sorted(L500_000, 10)
%timeit k_massimi_sorted(L1_000_000, 10)
```

```
40.4 µs ± 1.29 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
318 µs ± 618 ns per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
721 µs ± 5.9 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
4.44 ms ± 21.2 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
9.51 ms ± 89.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
57.3 ms ± 515 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
127 ms ± 660 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[206]:
```
%timeit k_massimi_lowmem(10, L1000)
%timeit k_massimi_lowmem(10, L5000)
%timeit k_massimi_lowmem(10, L10_000)
%timeit k_massimi_lowmem(10, L50_000)
%timeit k_massimi_lowmem(10, L100_000)
%timeit k_massimi_lowmem(10, L500_000)
%timeit k_massimi_lowmem(10, L1_000_000)
```

```
104 µs ± 83.3 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
437 µs ± 1.01 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
848 µs ± 1.34 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
4.1 ms ± 7.28 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
8.25 ms ± 16.3 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
40.8 ms ± 169 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
81.6 ms ± 173 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[211]:
```
xdata    = [1000, 5000, 10_000, 50_000, 100_000, 500_000, 1_000_000]
ydata_S  = [ 40.4, 318, 721, 4440, 9510, 57300, 127000 ]
ydata_LM = [104,   437, 848, 4100, 8250, 40800,  81600 ]
plt.figure()
plt.title("Lowmem with binary search (green) vs Sorted (orange):\n K=10␣
 ↪N=1000-1_000_000")
plt.scatter(xdata, ydata_S, color='tab:orange')
plt.plot(xdata, ydata_S, color='tab:orange')
plt.scatter(xdata, ydata_LM, color='tab:green')
plt.plot(xdata, ydata_LM, color='tab:green')
None
```



7.6 Wooclap finale