:::{list-table} :header-rows: 1 :align: center :widths: 90 10

- ▪ Fondamenti di programmazione

  Canale A-L - Prof. Sterbini

  AA 25-26 - Lezione 8

  - :::{image}attachment:5a837f06-202c-4c6e-9259-dd7a6dc01c97.png :width: 200px ::: :::

In [1…

```
from timeit import timeit # per misurare i tempi di esecuzione
```

RECAP

- ordinamenti complicati contrapposti
- funzioni anonime (lambda)
- lanciare errori con `raise`
- assert
- list comprehension
- inizio del problema dei K-massimi

Wooclap: FDPLEZ8

In [2…
```python
# SOLUZIONE 1
[ X+Y for X in "MiNnie" if X.isupper() for Y in "PlUtO" if Y.islower() ]
```

Out[2…
```
['Ml', 'Mt', 'Nl', 'Nt']
```

In [1…
```python
# SOLUZIONE 2
sorted( [ 1, 5, -23, 47, 12, -91 ], key=lambda numero: (-numero, numero) )
```

Out[1…
```
[47, 12, 5, 1, -23, -91]
```

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

Vediamo che parole sono state usate:

- **K**: input
- **lista L di valori numerici**: input
- **k massimi**: output
- **trovare i k massimi**: calcolo da eseguire

- **rappresentazione dei dati**?
  (in questo caso lo sappiamo)
  - INPUT: una lista L di numeri 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**?

Scegliamo ad esempio di modificare **L** distruttivamente

- se K è sbagliato diamo un **errore**
- **modifichiamo distruttivamente** L
- il risultato può essere in **qualsiasi ordine**

Strategia di soluzione

- Per estrarre i **K** massimi dalla lista **L**
    - controllo che gli argomenti siano validi altrimenti dò errore
    - 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

Digressione: come 'lanciare' errori

```
raise ClasseDellErrore(messaggio_di_errore)

# oppure (ma così lancio solo AssertionError)
assert condizione_normalmente_vera, messaggio_di_errore_se_falsa
```

Le asserzioni sono molto usate come controllo che durante l'esecuzione tutto vada bene.

NOTA: Possono essere disattivate quindi non vanno usate per controlli sui dati in input (che invece devono essere fatti sempre).

Per semplicità userò `assert`

In [3…
```python
# 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
    if not L:
        raise ValueError("L è vuota")
    # oppure
    assert len(L)>0,     "L è vuota"

    if not 0<k<=len(L):
        raise ValueError(f"K={k} non è compreso tra 1 e len(L)={len(L)}")
    # oppure
    assert 0<k<=len(L), f"K={k} non è compreso tra 1 e len(L)={len(L)}"

    risultato = []              # definisco la variabile risultato e la inizializzo = []
    for _ in range(k):          # ripeto per k volte (l'indice non mi interessa)
        M = estrai_massimo(L)   # estraggo un massimo da L eliminandolo dalla lista
        risultato.append(M)     # lo aggiungo al risultato
    return risultato            # torno i k massimi raccolti

    # oppure, usando una list-comprehension
    # return [ estrai_massimo(L) for _ in range(k) ]
```

In [4…
```python
# Se lo voglio scrivere con una list-comprehension ....
```

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

In [5…
```python
# per estrarre ed eliminare un massimo da una lista
def estrai_massimo(L):
    assert L, "La lista è vuota, non posso cercare il massimo"
    M = max(L)    # L NON deve essere vuota
    L.remove(M)   # sono sicuro che ci sia
```

```
    return M

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

Out[5… (89, [1, 5, 2, 2, 23])

In [6…
```
# costruiamo una lista di valori casuali (con ripetizioni)
from random import choices
help(choices)
L = list(choices(range(1_000_000), 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.

[64258, 393100, 722514, 870781, 466030, 144246, 298875, 719603, 859384, 167371]
```

In [7…
```
## vediamo quanto tempo impiega per diversi valori di K e per N=100_000
LL = list(choices(range(1_000_000), k=100_000))
None
help(timeit)
```

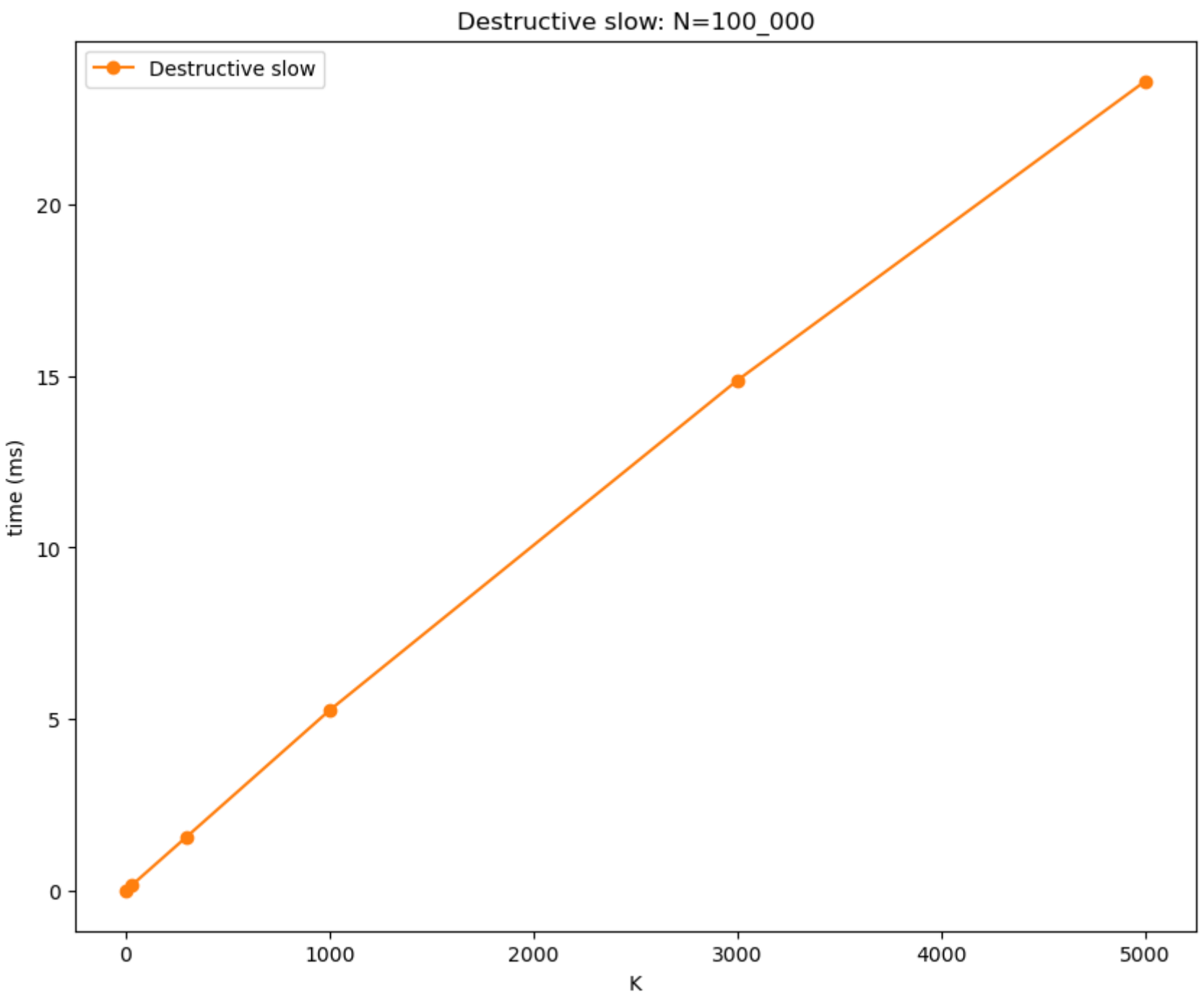
```
Help on function timeit in module timeit:

timeit(stmt='pass', setup='pass', timer=<built-in function perf_counter>, number=1000000, globals=None)
    Convenience function to create Timer object and call timeit method.
```

In [8…
```
%matplotlib inline
import matplotlib.pyplot as plt
xdata = [3, 30, 300, 1000, 3000, 5000]
ydata_slow = [timeit(lambda : k_massimi_distruttivo(LLL,X), number=5)
```

```
                for X in xdata
                for LLL in [LL.copy()]
               ]
```

In [9…

```
plt.figure(figsize=(10,8))
plt.title("Destructive slow: N=100_000")
plt.plot(xdata, ydata_slow,'o-', color='tab:orange')
plt.xlabel('K')
plt.ylabel('time (ms)')
plt.legend(['Destructive slow'])
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**

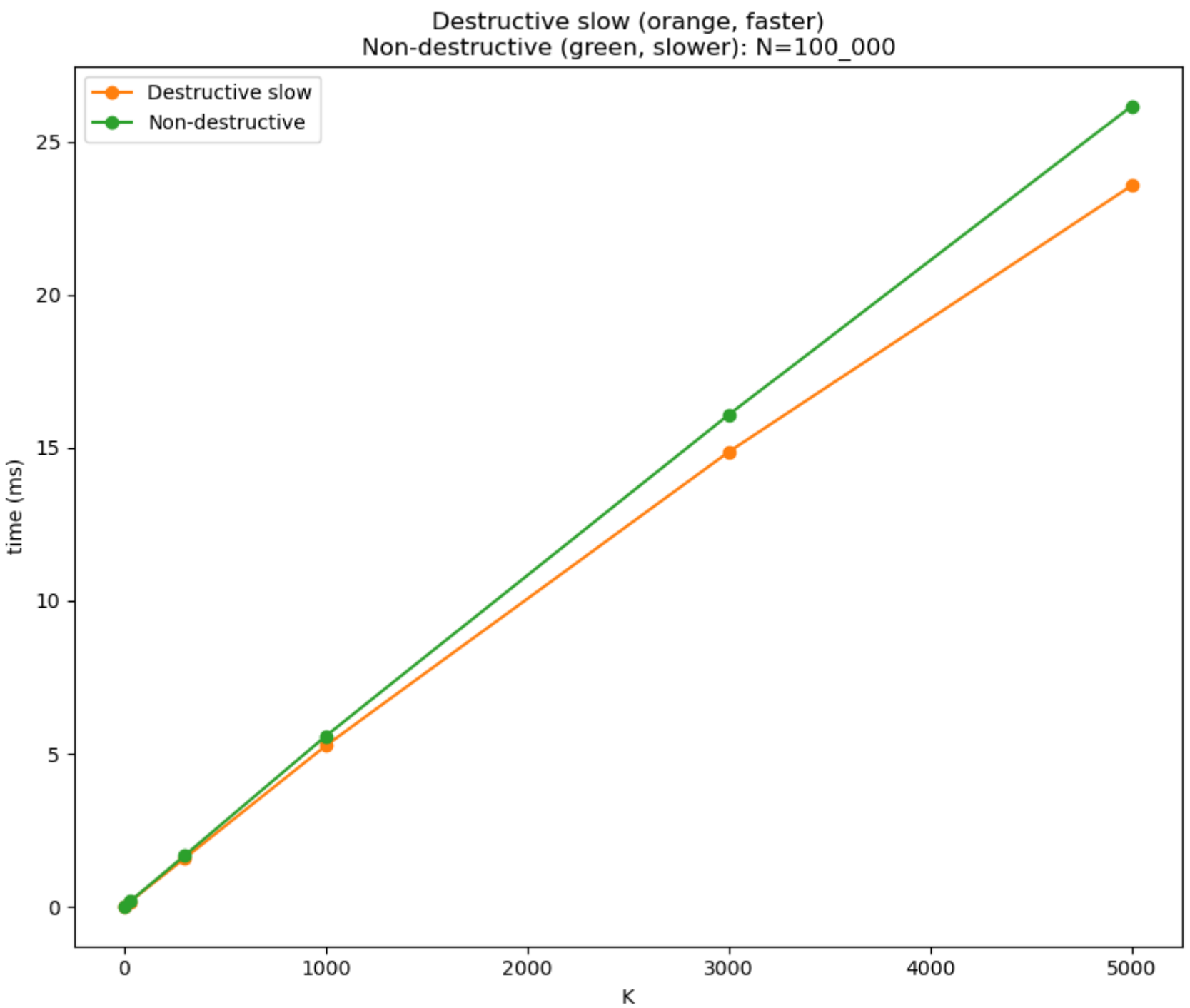
E se invece volessimo lasciare L invariata?

Basta copiarla

In [1…
```
# 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
```

In [1…
```
L = LL.copy()
xdata = [3, 30, 300, 1000, 3000, 5000]
ydata_slow2 = [timeit(lambda:k_massimi(L,X), number=5) for X in xdata] # timeit vuole una funzione senza
```

In [1…
```
plt.figure(figsize=(10,8))
plt.title("Destructive slow (orange, faster)\nNon-destructive (green, slower): N=100_000")
plt.plot(xdata, ydata_slow,  'o-', color='tab:orange')
plt.plot(xdata, ydata_slow2, 'o-', color='tab:green' )
plt.xlabel('K')
plt.ylabel('time (ms)')
plt.legend(['Destructive slow','Non-destructive'])
None
```



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

In [1…

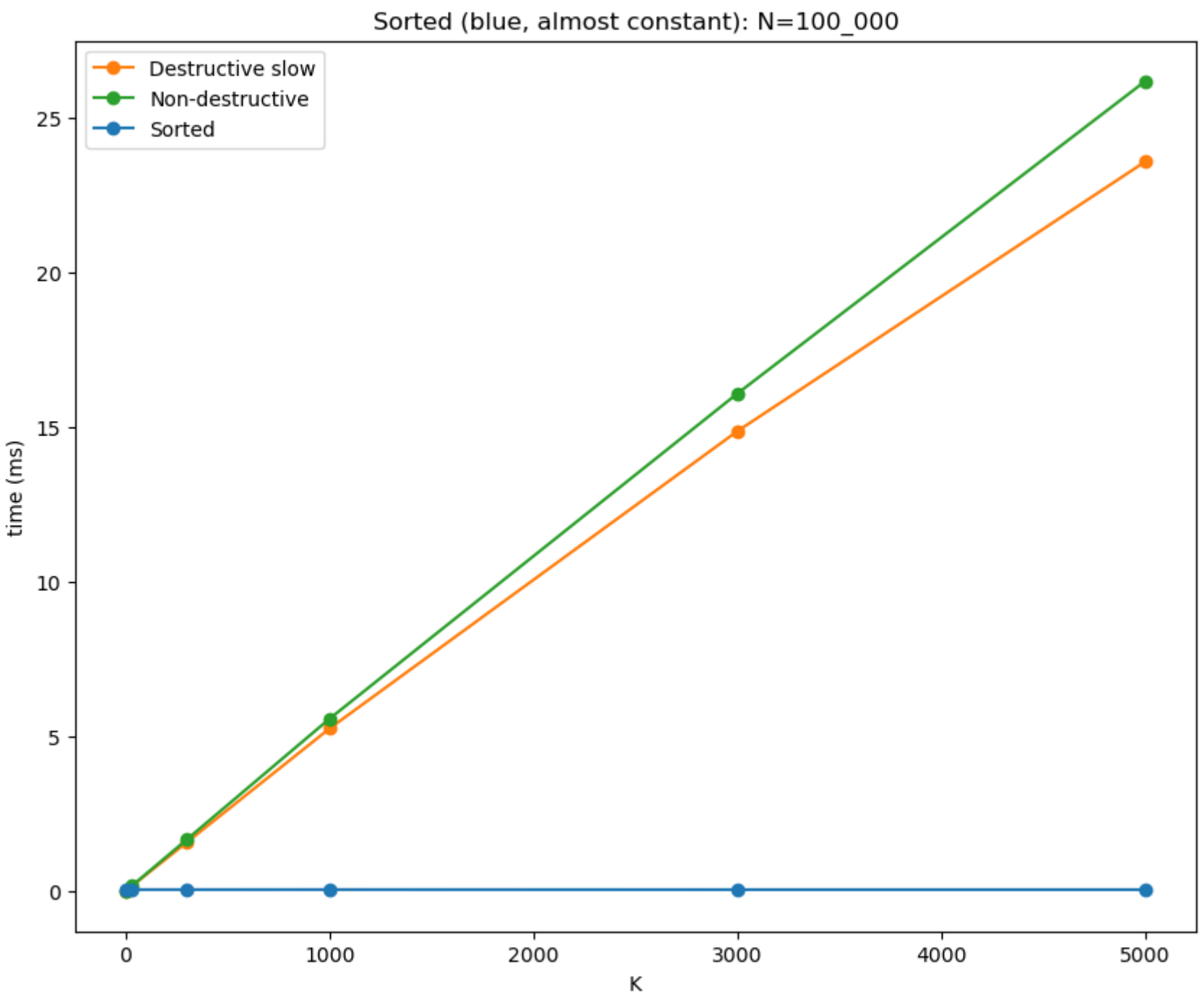
```
# Per estrarre i primi k massimi da L (non distruttivo)
def k_massimi_sorted(L, k):
    # controllo se k è valido                          tempo costante
    assert 0<k<=len(L)
    # ordino tutti i valori in ordine decrescente      tempo O(N*log(N))
    ordinata = sorted(L,reverse=True)
    # estraggo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

In [1…

```
xdata = [3, 30, 300, 1000, 3000, 5000]
ydata_sorted = [timeit(lambda:k_massimi_sorted(L,X), number=5) for X in xdata]
```

In [1…

```
plt.figure(figsize=(10,8))
plt.title("Sorted (blue, almost constant): N=100_000")
plt.plot(xdata, ydata_slow,   'o-', color='tab:orange')
plt.plot(xdata, ydata_slow2,  'o-', color='tab:green')
plt.plot(xdata, ydata_sorted, 'o-', color='tab:blue')
plt.xlabel('K')
plt.ylabel('time (ms)')
plt.legend(['Destructive slow','Non-destructive','Sorted'])
None
```



Vedete che stavolta il tempo quasi non è dipeso da **K**, infatti

- creare la lista ordinata impiega tempo proporzionale a $N*log_2(N)$

- estrarre i **K** massimi impiega tempo proporzionale a **K**

Quindi il tempo totale è proporzionale a **$K + N*log_2(N)$**

Ovvero, se **$K \ll N$**, ignoriamo **K** e il contributo principale è **$N*log_2(N)$**

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

In [1…

```python
def k_massimi_lowmem(K, L):
    massimi = []
    for X in L:                          # per N volte
        update_massimi(massimi, K, X)    # che tempo impiega ?
    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** ho 3 casi:

1. se ho meno di **K** valori
   - allora aggiungo **X** ai massimi correnti
2. altrimenti sono K, e se **X** è minore o uguale al più piccolo dei massimi raccolti
   - allora lo posso tranquillamente ignorare
3. altrimenti è un valore da ricordare

- elimino il minimo dai massimi raccolti
- aggiungo **X** ai massimi

In [1…
```
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
```

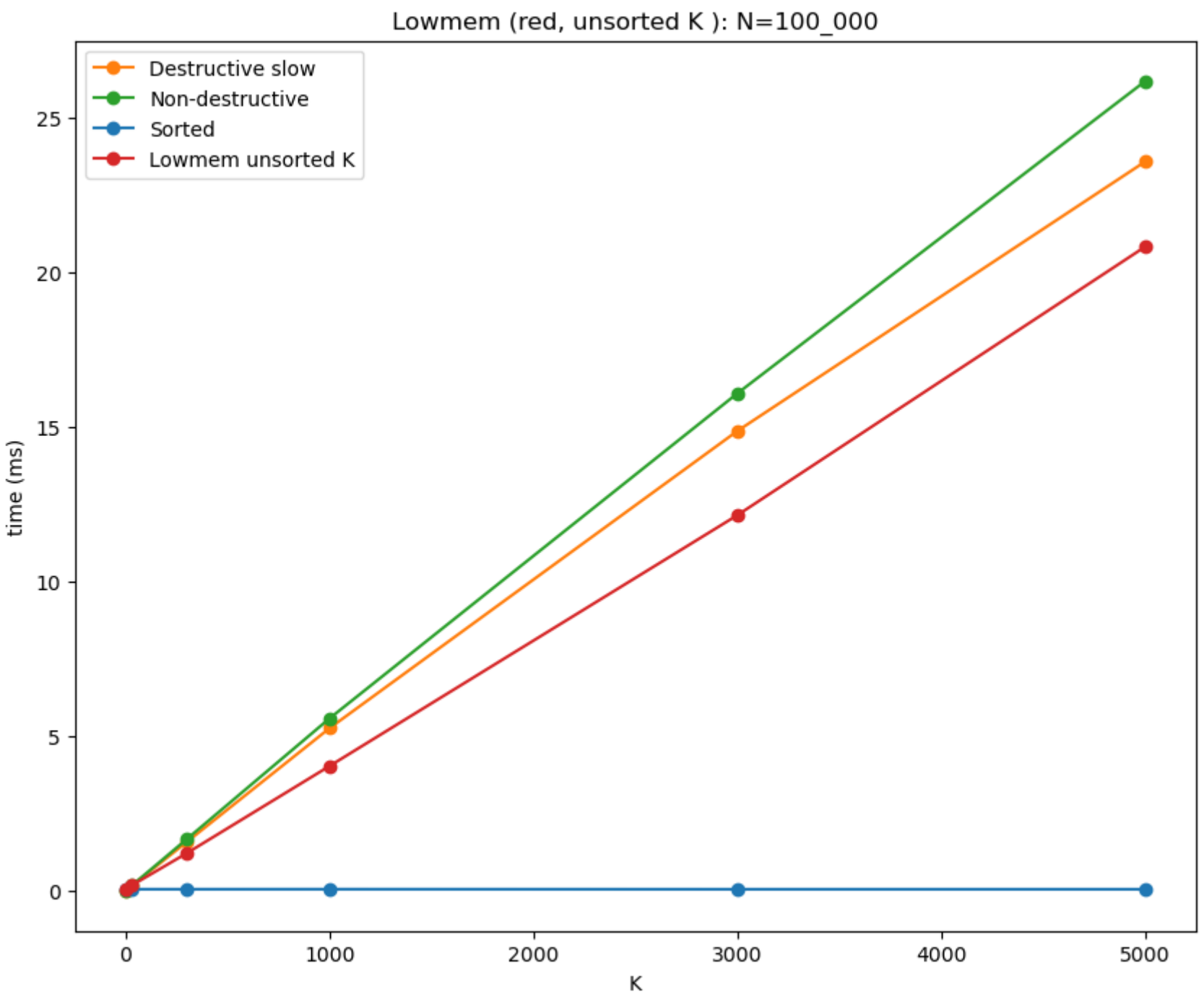
In [1…
```
# Esempio di aggiornamento
massimi = [12, 45, 67, 1, 90, -2, 17]
print('i massimi iniziali sono\t\t\t\t\t',massimi)
update_massimi(massimi, 10, 51)
print('se ho meno di K=10 massimi aggiungo X=51\t\t', massimi)
update_massimi(massimi, 8, -29)
print('se X=-29 è minore del minimo lo ignoro\t\t\t',massimi)
update_massimi(massimi, 8, 23)
print('se X=23 è maggiore del minimo rimpiazzo il minimo\t',massimi)
```

```
i massimi iniziali sono                                 [12, 45, 67, 1, 90, -2, 17]
se ho meno di K=10 massimi aggiungo X=51                [12, 45, 67, 1, 90, -2, 17, 51]
se X=-29 è minore del minimo lo ignoro                  [12, 45, 67, 1, 90, -2, 17, 51]
se X=23 è maggiore del minimo rimpiazzo il minimo       [12, 45, 67, 1, 90, 17, 51, 23]
```

In [1…
```
L = LL.copy()
xdata = [3, 30, 300, 1000, 3000, 5000]
ydata_lowmem1 = [timeit(lambda:k_massimi_lowmem(X,L), number=5) for X in xdata]
```

In [2…
```
plt.figure(figsize=(10,8))
plt.title("Lowmem (red, unsorted K ): N=100_000")
plt.plot(xdata, ydata_slow,    'o-', color='tab:orange')
```

```
plt.plot(xdata, ydata_slow2,   'o-', color='tab:green')
plt.plot(xdata, ydata_sorted,  'o-', color='tab:blue')
plt.plot(xdata, ydata_lowmem1, 'o-', color='tab:red')
plt.xlabel('K')
plt.ylabel('time (ms)')
plt.legend(['Destructive slow','Non-destructive','Sorted','Lowmem unsorted K'])
None
```



miglioriamo la gestione dei **K** massimi

che succederebbe se li mantenessimo ordinati?

Per aggiornare i **K** massimi

- trovare il minimo diventa costante (è sempre in fondo)
- eliminare il minimo lo stesso (lavorare sull'ultimo è veloce)
- 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**)

In [2…

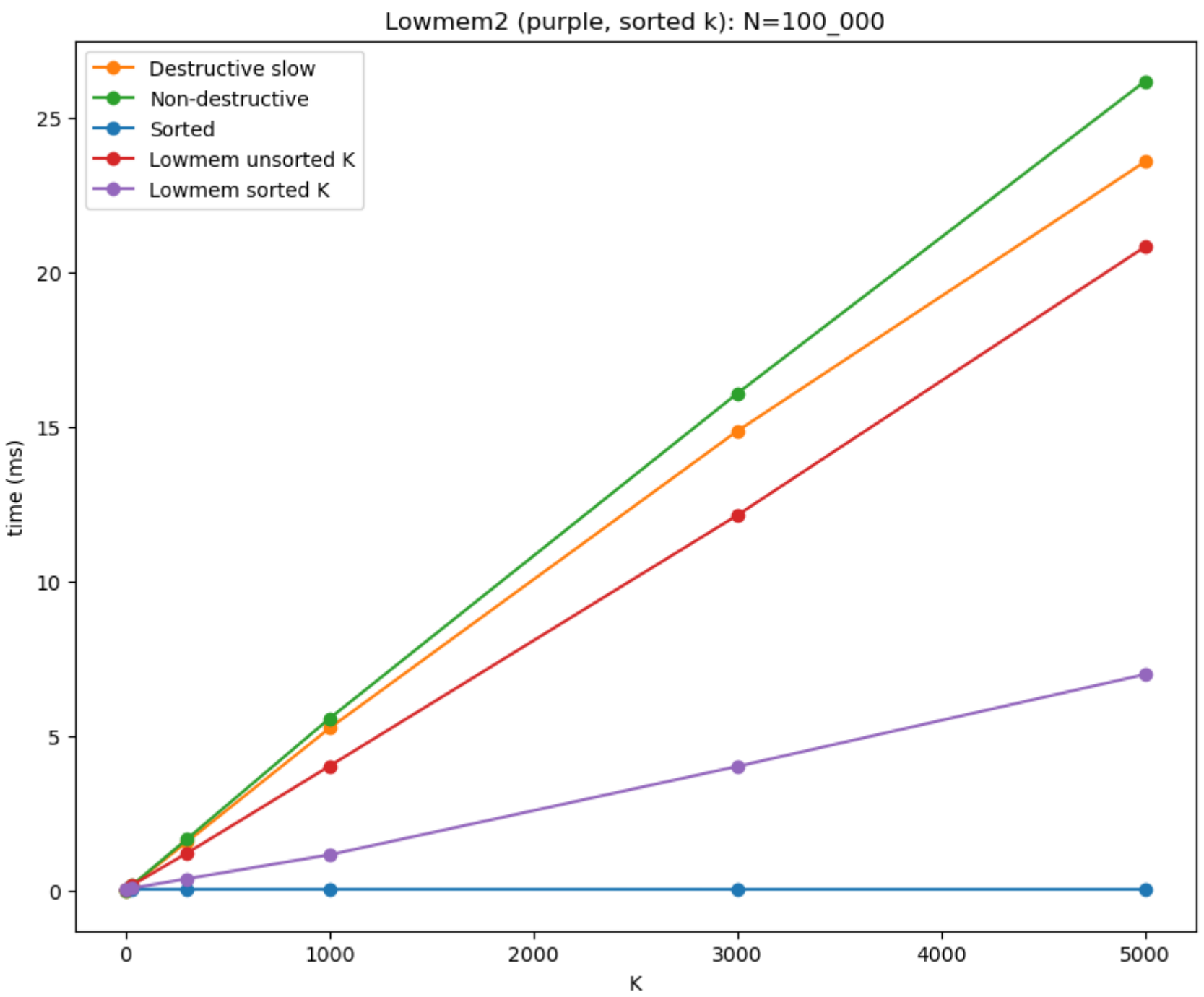
```
## Proviamo quello più semplice da scrivere
def update_massimi(massimi, K, X):
    massimi.append(X)                # tempo costante
    massimi.sort(reverse=True)       # tempo K*log(K)
    massimi[K:] = []                 # elimino i valori oltre la posizione K se esistono, tempo costante
```

In [2…

```
xdata = [3, 30, 300, 1000, 3000, 5000]
ydata_lowmem2 = [timeit(lambda:k_massimi_lowmem(X,L), number=5) for X in xdata]
```

In [2…

```
plt.figure(figsize=(10,8))
plt.title("Lowmem2 (purple, sorted k): N=100_000")
plt.plot(xdata, ydata_slow,    'o-', color='tab:orange')
plt.plot(xdata, ydata_slow2,   'o-', color='tab:green')
plt.plot(xdata, ydata_sorted,  'o-', color='tab:blue')
plt.plot(xdata, ydata_lowmem1, 'o-', color='tab:red')
plt.plot(xdata, ydata_lowmem2, 'o-', color='tab:purple')
plt.xlabel('K')
plt.ylabel('time (ms)')
plt.legend(['Destructive slow','Non-destructive','Sorted','Lowmem unsorted K','Lowmem sorted K'])

None
```



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

In [2…

```
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)
```

Ricerca binaria:

- Per cercare un elemento X in una lista ordinata decrescente:
  - se non ci sono elementi torno la posizione in cui sto cercando
  - altrimenti
    - prendo l'elemento centrale Y
    - se è quello cercato ho trovato la posizione desiderata
    - altrimenti se X<Y
      - devo cercare a destra di Y
      - altrimenti devo cercare a sinistra di Y

Ricerca binaria: divido in due la ricerca ad ogni passo

Per cercare dove inserire un elemento X in una lista **ORDINATA** decrescente

- inizializzo **inizio** e **fine** agli indici del primo e ultimo elemento (zona di ricerca)
- 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 X è MINORE del valore centrale
    - 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 corrente

In [3…

```
def ricerca_binaria(Lista, Valore, debug=False):
    inizio = 0
    fine = len(Lista)-1
    while inizio <= fine:                  # se ci sono elementi da esaminare
        centrale = (inizio + fine)//2      # calcolo la posizione centrale
        valore_centrale = Lista[centrale]  # e leggo l'elemento centrale
        if debug:
            print('cerco', Valore,'inizio', inizio, 'fine', fine,
                  'centrale', centrale, 'contiene', valore_centrale, sep='\t')
        if Valore == valore_centrale:      # se l'ho trovato
            return centrale                # torno la posizione centrale
        elif Valore < valore_centrale:     # se il valore cercato è minore del centro
            inizio = centrale+1            # sposto l'inizio alla destra e continuo
        else:
            fine = centrale-1              # altrimenti sposto la fine al centro e continuo
    return inizio                          # se ho finito il ciclo in 'inizio' ho la posizione
```

In [4…

```
# Esempio
ordinati = list(range(100,-1,-2))       # lista decrescente di valori pari
```

```
print(ordinati)
pos = ricerca_binaria(ordinati, 33, True)
ordinati.insert(pos,33)
print(ordinati)
```

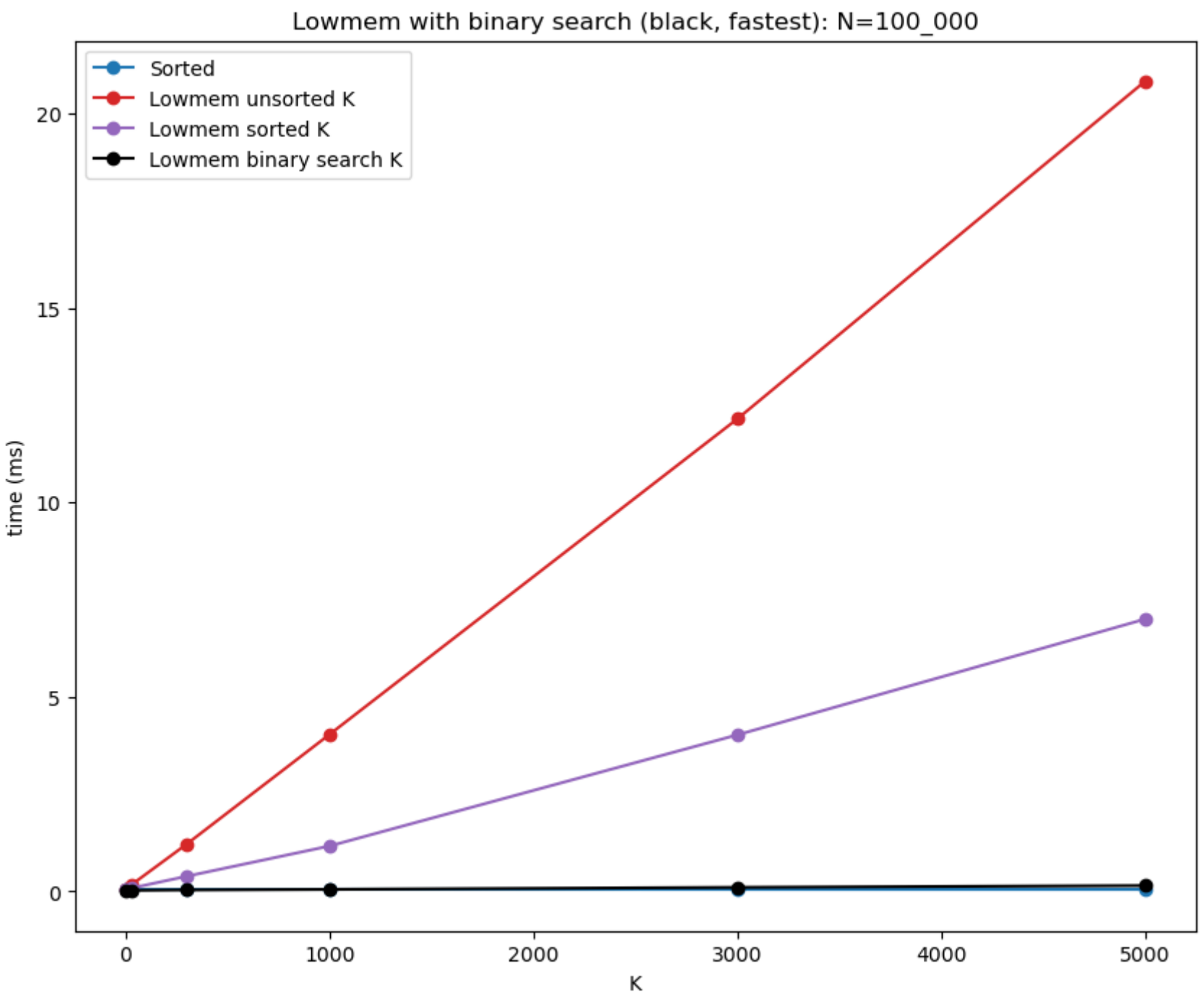
```
[100, 98, 96, 94, 92, 90, 88, 86, 84, 82, 80, 78, 76, 74, 72, 70, 68, 66, 64, 62, 60, 58, 56, 54, 52, 50
, 48, 46, 44, 42, 40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14, 12, 10, 8, 6, 4, 2, 0]
cerco   33      inizio  0       fine    50      centrale        25      contiene        50
cerco   33      inizio  26      fine    50      centrale        38      contiene        24
cerco   33      inizio  26      fine    37      centrale        31      contiene        38
cerco   33      inizio  32      fine    37      centrale        34      contiene        32
cerco   33      inizio  32      fine    33      centrale        32      contiene        36
cerco   33      inizio  33      fine    33      centrale        33      contiene        34
[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]
```

In [2…

```
xdata = [3, 30, 300, 1000, 3000, 5000]
ydata_lowmem3 = [timeit(lambda:k_massimi_lowmem(X,L), number=5) for X in xdata]
```

In [2…

```
plt.figure(figsize=(10,8))
plt.title("Lowmem with binary search (black, fastest): N=100_000")
#plt.plot(xdata, ydata_slow,    'o-', color='tab:orange')
#plt.plot(xdata, ydata_slow2,   'o-', color='tab:green')
plt.plot(xdata, ydata_sorted,  'o-', color='tab:blue')
plt.plot(xdata, ydata_lowmem1, 'o-', color='tab:red')
plt.plot(xdata, ydata_lowmem2, 'o-', color='tab:purple')
plt.plot(xdata, ydata_lowmem3, 'o-', color='black')
plt.xlabel('K')
plt.ylabel('time (ms)')
plt.legend(['Sorted','Lowmem unsorted K','Lowmem sorted K','Lowmem binary search K'])

None
```



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	$log(N)$	$N*log(N)$
10	circa 3	circa 30
100	circa 7	circa 700
1000	circa 10	circa 10_000
1_000_000	circa 20	circa 20_000_000

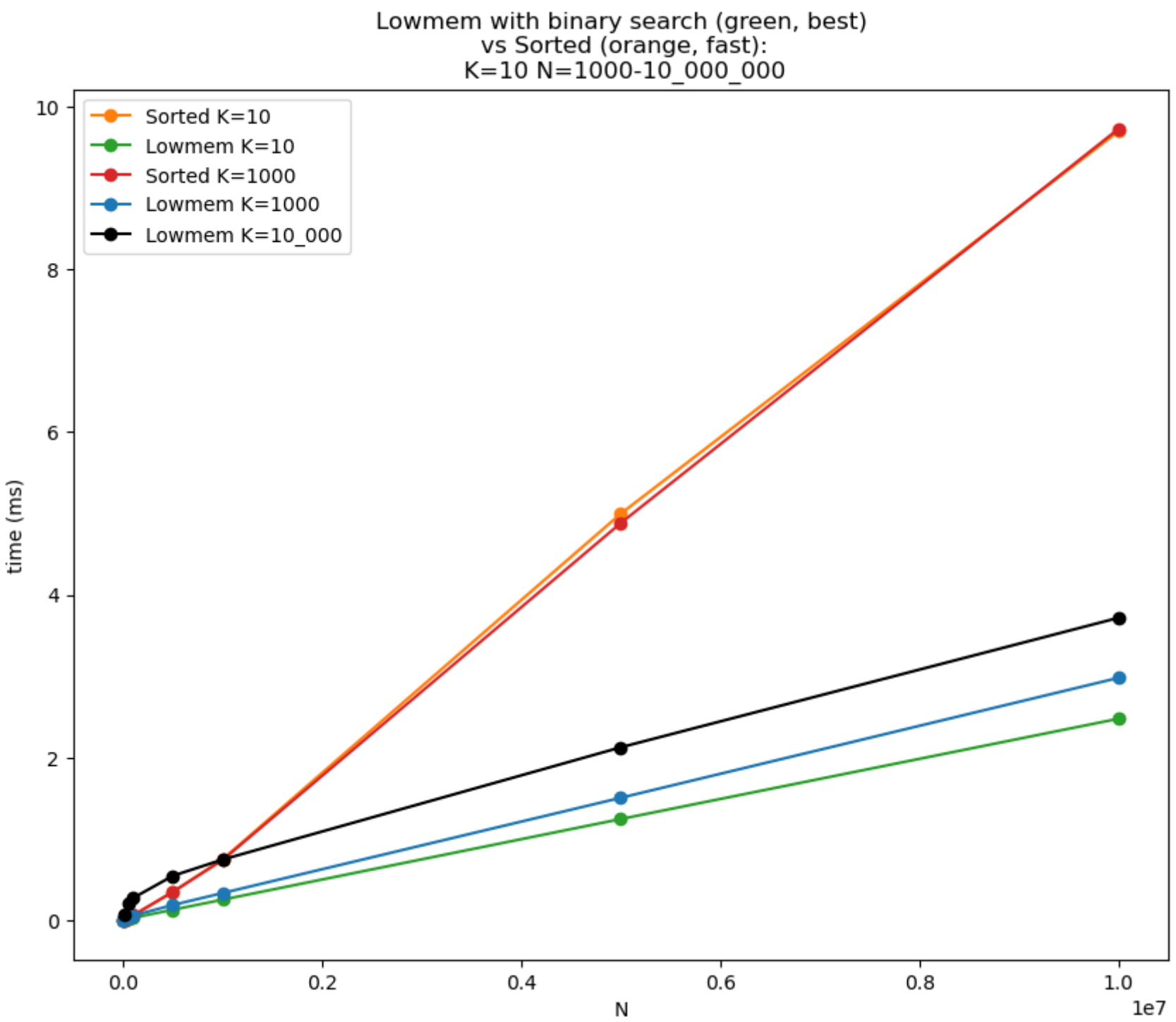
- se manteniamo ordinati solo i **K** massimi impieghiamo nel caso peggiore tempo **$N*(K+log(K)) = K*N$** (si ignora log(K))

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

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

In [2…
```
xdata    = [1000, 5000, 10_000, 50_000, 100_000, 500_000, 1_000_000, 5_000_000, 10_000_000]
liste    = [list(choices(range(10000000), k=N)) for N in xdata]
# K = 10
ydata_S  = [ timeit(lambda:k_massimi_sorted(L,10), number=5)    for L in liste ]
ydata_LM = [ timeit(lambda:k_massimi_lowmem(10,L), number=5)    for L in liste ]
# K = 1000
ydata_S_K= [ timeit(lambda:k_massimi_sorted(L,1000), number=5)  for L in liste  ]
ydata_LM_K=[ timeit(lambda:k_massimi_lowmem(1000,L), number=5)  for L in liste  ]
# K = 10000
ydata_S_XK= [ timeit(lambda:k_massimi_sorted(L,10000), number=5)  for L in liste[2:]  ]
ydata_LM_XK=[ timeit(lambda:k_massimi_lowmem(10000,L), number=5)  for L in liste[2:]  ]
```

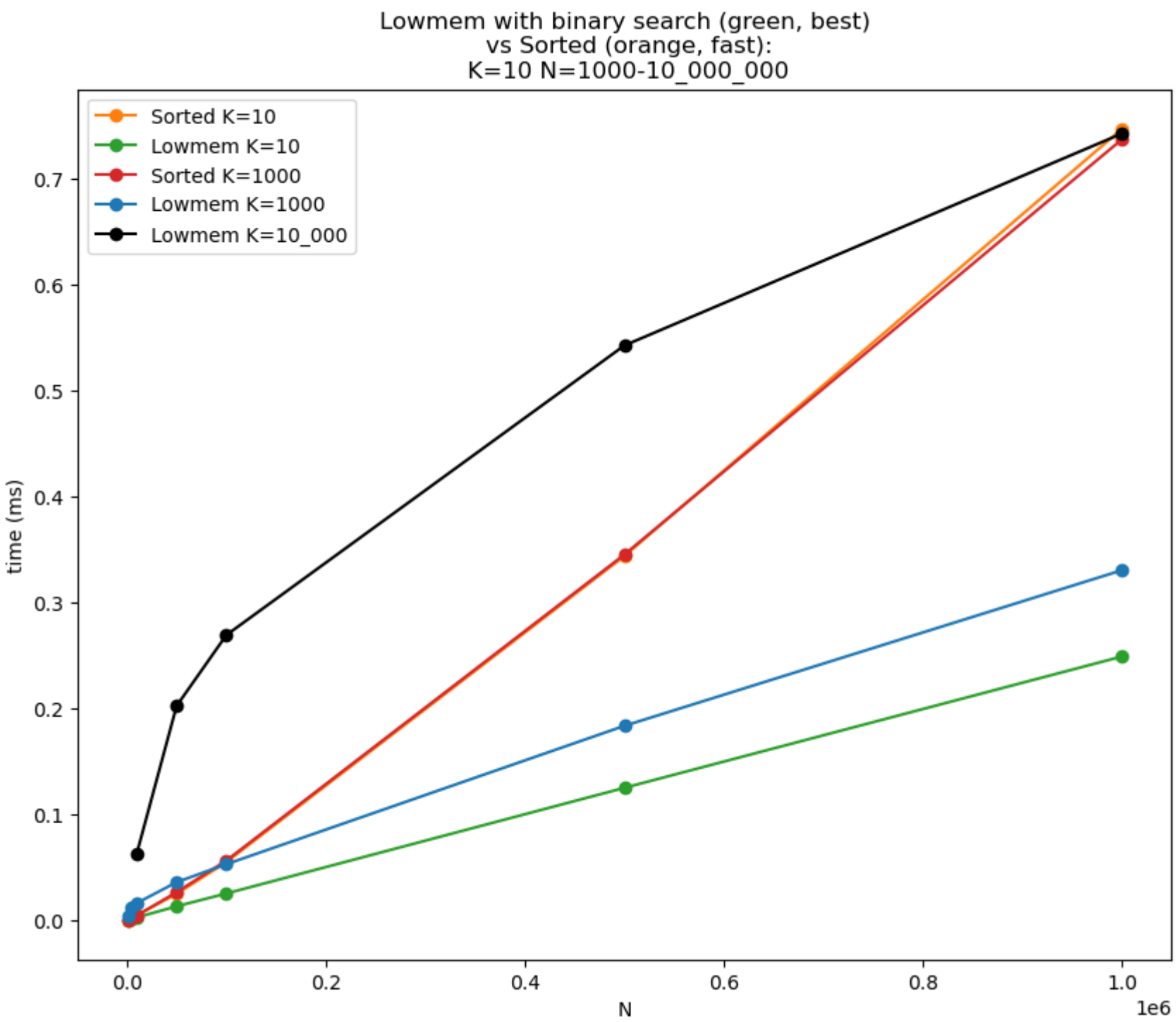
In [3…
```
plt.figure(figsize=(10,8))
plt.title("Lowmem with binary search (green, best)\n vs Sorted (orange, fast):\n K=10 N=1000-10_000_000'
plt.plot(xdata,     ydata_S,     'o-', color='tab:orange')
plt.plot(xdata,     ydata_LM,    'o-', color='tab:green')
plt.plot(xdata,     ydata_S_K,   'o-', color='tab:red')
plt.plot(xdata,     ydata_LM_K,  'o-', color='tab:blue')
plt.plot(xdata[2:], ydata_LM_XK, 'o-', color='black')
plt.xlabel('N')
plt.ylabel('time (ms)')
plt.legend(['Sorted K=10', 'Lowmem K=10','Sorted K=1000','Lowmem K=1000','Lowmem K=10_000'])
None
```



Da cui sembra che la soluzione lowmem ordinata sia sempre la migliore

In [3…

```
plt.figure(figsize=(10,8))
plt.title("Lowmem with binary search (green, best)\n vs Sorted (orange, fast):\n K=10 N=1000-10_000_000"
plt.plot(xdata[:-2],  ydata_S[:-2],     'o-', color='tab:orange')
plt.plot(xdata[:-2],  ydata_LM[:-2],    'o-', color='tab:green')
plt.plot(xdata[:-2],  ydata_S_K[:-2],   'o-', color='tab:red')
plt.plot(xdata[:-2],  ydata_LM_K[:-2],  'o-', color='tab:blue')
plt.plot(xdata[2:-2], ydata_LM_XK[:-2], 'o-', color='black')
plt.xlabel('N')
plt.ylabel('time (ms)')
plt.legend(['Sorted K=10', 'Lowmem K=10','Sorted K=1000','Lowmem K=1000','Lowmem K=10_000'])
None
```



Se guardiamo però meglio i dati per N più piccolo fino a 1_000_000

- per K=10 la soluzione con memoria ridotta è più veloce della soluzione ordinando tutta la lista
- per K=1000 la differenza è minore, come previsto
- per K>10_000 la soluzione con memoria ridotta è più lenta della soluzione ordinando tutta la lista

Tutto dipende dai due valori N e K

- se K è molto più piccolo di log(N) conviene la soluzione con memoria ridotta
- se K è comparabile con log(N) conviene ordinare tutta la lista

Qualche micro nozione di complessità temporale dei programmi: la notazione O

$O(f(n))$: nel caso peggiore il tempo impiegato è ***"simile"*** alla funzione **$f(n)$**
(con **n** che è la dimensione dei dati in input)

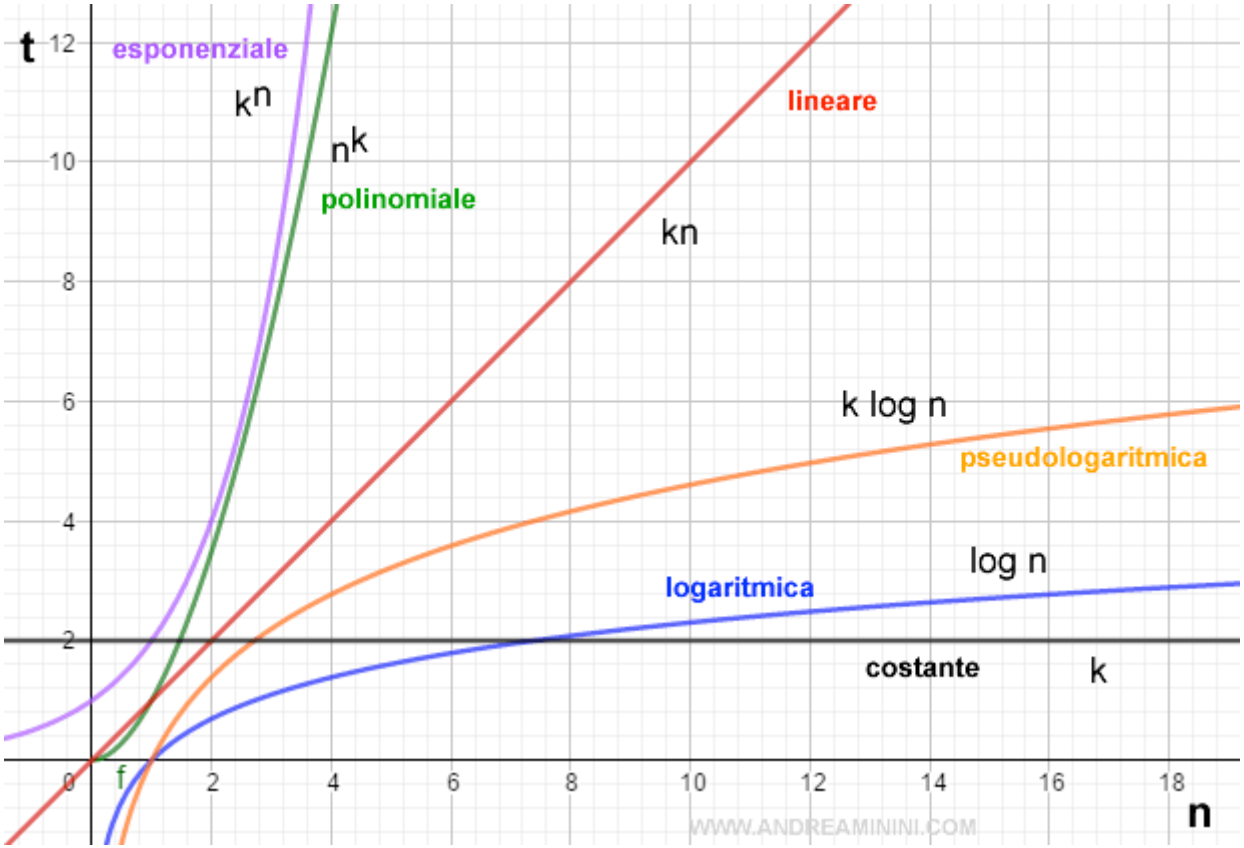
ovvero "si comporta" o "cresce" come la funzione **$f(n)$**

Dal più veloce al più lento:

notazione O	che significa tempo ...	ovvero?
$O(1)$	costante	non dipende dalle dimensioni dell'input
$O(log(N))$	logaritmico rispetto all'input	per ogni raddoppio di N il tempo aument se $N*1000$ allora $tempo*10$ perchè $log(1000) \tilde= 10$
$O(N)$	proporzionale all'input	se N raddoppia il tempo raddoppia
$O(N * log(N))$	proporzionale all'input * il suo logaritmo	se $N*1000$ il tempo aumenta di $1000*1 volte perchè $log(1000) \tilde= 10$
$O(N^2)$	quadratico	se N raddoppia il tempo quadruplica se $N*10$ allora $tempo*100$

notazione O	che significa tempo ...	ovvero?
$O(2^N)$	esponenziale	se N aumenta di 1, il tempo raddoppia se N aumenta di 10 allora $tempo*1000$

- $log(N)=log_2(N)$ numero di cifre binarie per rappresentare N



Come si comportano somme e prodotti degli ordini di grandezza

Visto che stiamo cercando **il caso peggiore**:

Se **N** è la dimensione dell'input da elaborare:

la formula	diventa	perchè
$O(costante * F(N))$	$O(F(N))$	ignoriamo i fattori cos

la formula	diventa	perchè
$O(F(N)) + O(G(N))$	$O(max(F(N),G(N)))$	vogliamo il caso pegg
$O(F(N)) * O(G(N))$	$O(F(N)*G(N))$	si moltiplicano

Esempi:

la formula		diventa	perchè
$O(5*n)$	==>	$O(n)$	si ignorano i fattori costanti
$O(log(n)) + O(1)$	==>	$O(log(n))$	il logaritmo cresce mentre una cos
$O(log(n)) + O(n)$	==>	$O(n)$	il logaritmo cresce più lentament
$O(n)*O(n)$	==>	$O(n^2)$	$n*n = n^2$
$O(n)*O(log(n))+O(n)$	==>	$O(n*log(n))$	n*log cresce più rapidamente

Complessità di estrarre i k massimi da N valori

Dobbiamo ripetere **$O(n)$** volte (per tutti i valori letti)

- aggiorno la lista di **k** migliori elementi
  - tempo $O(k)$ se **non** tengo i k migliori ordinati
  - oppure $O(k*log(k))$ se li tengo ordinati con sorted

Sembrerebbe inutile tenerli ordinati!

Avevo dato un suggerimento per migliorare

- trovare in tempo $O(log(k))$ dove inserire X
- inserire X (purtroppo ancora in tempo $O(k)$)

E questo ci riporta di nuovo a $O(k) + log(k) = O(k)$ per aggiornare i k elementi

Ci vorrebbe una struttura dati con inserimento, cancellazione e minimo più veloce in tempo $O(log(k))$

SOLUZIONE: from sortedcontainers import SortedList

- $O(log(k))$ inserimento di un nuovo valore
- $O(log(k))$ lettura minimo (elemento a indice 0)
- $O(log(k))$ cancellazione del minimo

Risultato finale per i k-massimi: tempo **$O(n*log(k))$**

In [3…

```
from sortedcontainers import SortedList
help(SortedList.add)
help(SortedList.__delitem__) # usato da 'del massimi[0]' per eliminare il minimo
help(SortedList.__getitem__) # usato da 'massimi[0]'     per leggere il minimo
```

```
Help on function add in module sortedcontainers.sortedlist:

add(self, value)
    Add `value` to sorted list.

    Runtime complexity: `O(log(n))` -- approximate.

    >>> sl = SortedList()
    >>> sl.add(3)
    >>> sl.add(1)
    >>> sl.add(2)
    >>> sl
    SortedList([1, 2, 3])

    :param value: value to add to sorted list

Help on function __delitem__ in module sortedcontainers.sortedlist:

__delitem__(self, index)
    Remove value at `index` from sorted list.

    ``sl.__delitem__(index)`` <==> ``del sl[index]``

    Supports slicing.

    Runtime complexity: `O(log(n))` -- approximate.

    >>> sl = SortedList('abcde')
    >>> del sl[2]
    >>> sl
    SortedList(['a', 'b', 'd', 'e'])
    >>> del sl[:2]
    >>> sl
    SortedList(['d', 'e'])

    :param index: integer or slice for indexing
    :raises IndexError: if index out of range
```

```
Help on function __getitem__ in module sortedcontainers.sortedlist:

__getitem__(self, index)
    Lookup value at `index` in sorted list.

    ``sl.__getitem__(index)`` <==> ``sl[index]``

    Supports slicing.

    Runtime complexity: `O(log(n))` -- approximate.

    >>> sl = SortedList('abcde')
    >>> sl[1]
    'b'
    >>> sl[-1]
    'e'
    >>> sl[2:5]
    ['c', 'd', 'e']

    :param index: integer or slice for indexing
    :return: value or list of values
    :raises IndexError: if index out of range
```

In [3…

```python
from sortedcontainers import SortedList
def aggiorna_k_massimi_SC(massimi:SortedList, X:int, k:int) -> None:
    if len(massimi) < k:
        massimi.add(X)      # O(log(k))
    elif massimi[0] < X:    # O(log(k))
        del massimi[0]      # O(log(k))
        massimi.add(X)      # O(log(k))

# quindi il tempo nel caso peggiore è O(log(k))
```

In [3…

```python
# cerchiamo il caso peggiore: la sequenza di valori crescente
# avrà i k massimi in fondo e dovrà sempre aggiornare i k massimi
def k_massimi_SC_crescente(N:int, k:int) -> SortedList:
    massimi : SortedList = SortedList()      # invece di una lista usiamo un SortedContainer
```

```
    for X in range(N):                          # il caso peggiore è la sequenza crescente, ripetuto N vo
        aggiorna_k_massimi_SC(massimi, X, k)  # O(log(k))
    return massimi

# quindi il tempo peggiore è O(log(k)*N)
```

In [3…

```
# Riprendiamo le implementazioni della lezione 7 e confrontiamo i tempi
def k_massimi_sorted_binary(N, k):          # O(N*k)
    massimi = []
    for X in range(N):          # il caso peggiore è la sequenza crescente, ripetuto N volte
        aggiorna_k_massimi_binary(massimi, X, k)  # O(k)
    return massimi

def aggiorna_k_massimi_binary(massimi, X, k): # O(k)
    if len(massimi) < k:
        pos = ricerca_binaria(massimi, X)  # O(log(k))
        massimi.insert(pos,X)              # O(k)  <- slowest
    elif massimi[-1] < X:                  # O(1)
        del massimi[-1]                    # O(1)
        pos = ricerca_binaria(massimi, X)  # O(log(k))
        massimi.insert(pos,X)              # O(k)  <- slowest

def ricerca_binaria(Lista, Valore):        # O(log(k))
    inizio = 0
    fine = len(Lista)-1
    while inizio <= fine:                  # se ci sono elementi da esaminare
        centrale = (inizio + fine)//2      # calcolo la posizione centrale
        valore_centrale = Lista[centrale]  # e leggo l'elemento centrale
        if Valore == valore_centrale:      # se l'ho trovato
            return centrale                # torno la posizione centrale
        elif Valore < valore_centrale:     # se il valore cercato è minore del centro
            inizio = centrale+1            # sposto l'inizio alla destra e continuo
        else:
            fine = centrale-1              # altrimenti sposto la fine al centro e continuo
    return inizio                          # se ho finito il ciclo in 'inizio' ho la posizione
```

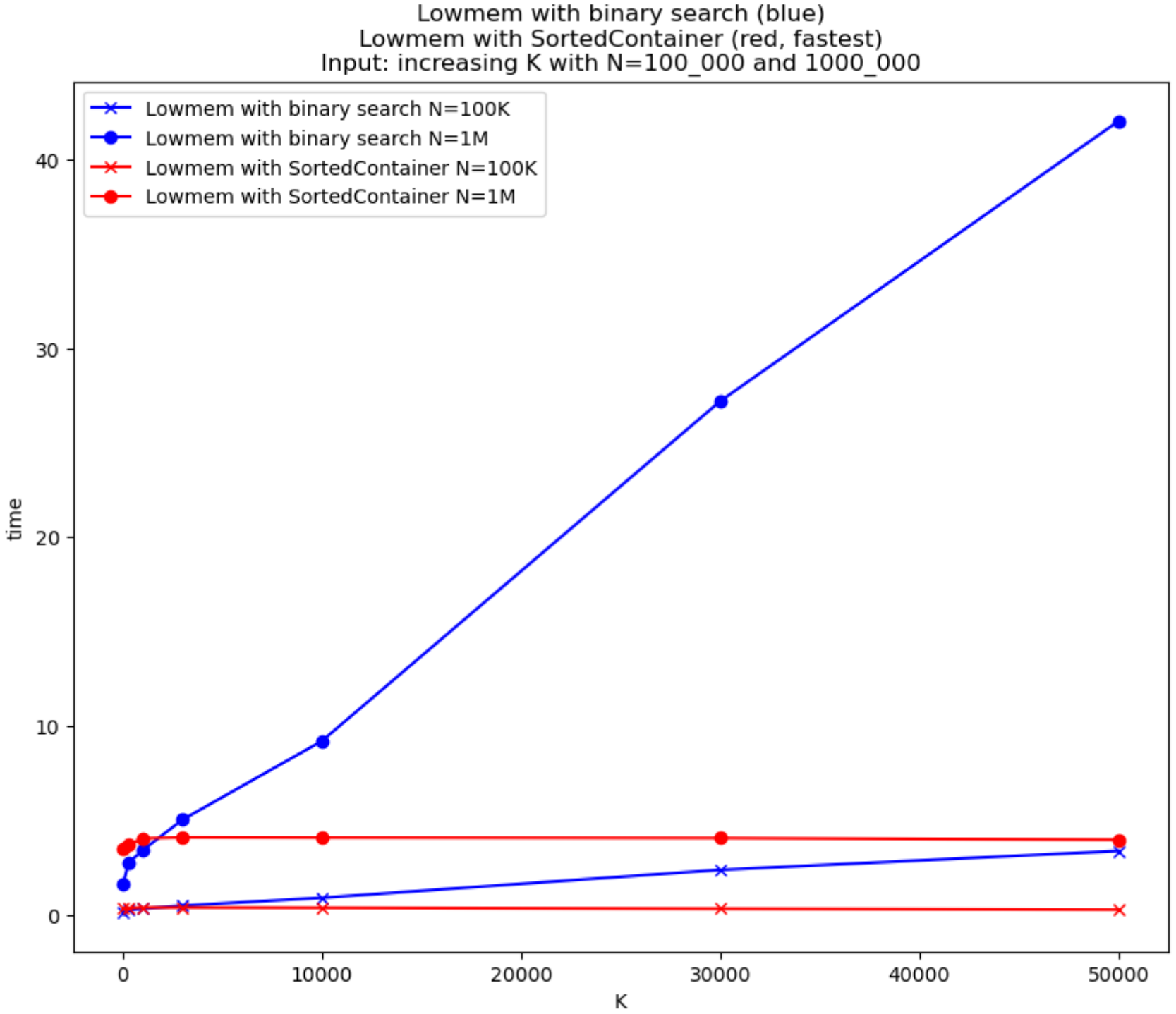
In [3…

```
%matplotlib inline
```

```
import matplotlib.pyplot as plt
from timeit import timeit
xdata                       = [ 30, 300, 1_000, 3_000, 10_000, 30_000, 50_000]
ydata_lowmem_bin_100K  : list[float] = [timeit(lambda:k_massimi_sorted_binary( 100_000,X), number=5) for
ydata_lowmem_bin_1M    : list[float] = [timeit(lambda:k_massimi_sorted_binary(1000_000,X), number=5) for
ydata_lowmem_SC_100K   : list[float] = [timeit(lambda:k_massimi_SC_crescente(  100_000,X), number=5) for
ydata_lowmem_SC_1M     : list[float] = [timeit(lambda:k_massimi_SC_crescente( 1000_000,X), number=5) for
```

In [3…

```
plt.figure(figsize=(10,8))
plt.xlabel('K')
plt.ylabel('time')
plt.title("""Lowmem with binary search (blue)
Lowmem with SortedContainer (red, fastest)
Input: increasing K with N=100_000 and 1000_000""")
plt.plot(xdata, ydata_lowmem_bin_100K,  'bx-')
plt.plot(xdata, ydata_lowmem_bin_1M,    'bo-')
plt.plot(xdata, ydata_lowmem_SC_100K,   'rx-')
plt.plot(xdata, ydata_lowmem_SC_1M,     'ro-')

plt.legend(['Lowmem with binary search N=100K',   'Lowmem with binary search N=1M',
            'Lowmem with SortedContainer N=100K', 'Lowmem with SortedContainer N=1M'])
None
```



Da cui si vede che:

- per N=100_000 le due soluzioni si equivalgono
- per N=1_000_000 la soluzione con SortedContainer è molto più veloce