# lezione07

October 16, 2023

## 1 Fondamenti di Programmazione

**Andrea Sterbini**

lezione 7 - 16 ottobre 2023

## 2 RECAP: COME analizzare un problema?

- 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**

### 2.1 RECAP: k massimi di N numeri

Se **tengo tutti i dati in memoria** - versione modifica distruttiva (tempo $O(N * k)$) - versione copia e modifica (tempo $O(N * k)$) - versione ordinamento e estrazione (tempo $O(N * log(N))$

## 3 Qualche micro nozione di complessità temporale dei programmi

$O(f(n))$ : 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: - $O(1)$ : tempo costante (non dipende dall'input) - $O(log(N))$ : tempo logaritmico rispetto all'input (per ogni raddoppio di N il tempo aumenta di 1) - $O(N)$ : tempo proporzionale all'input (se N raddoppia il tempo raddoppia) - $O(N * log(N))$ : tempo proporzionale all'input * il suo logaritmo (p.es. sort, se N*1000 il tempo aumenta di 10_000)* - $O(N^2)$ *: tempo*

*quadratico (se N raddoppia il tempo*4) - $O(2^N)$ : tempo esponenziale (se N aumenta di 1, il tempo raddoppia)

logaritmo = numero di cifre binarie

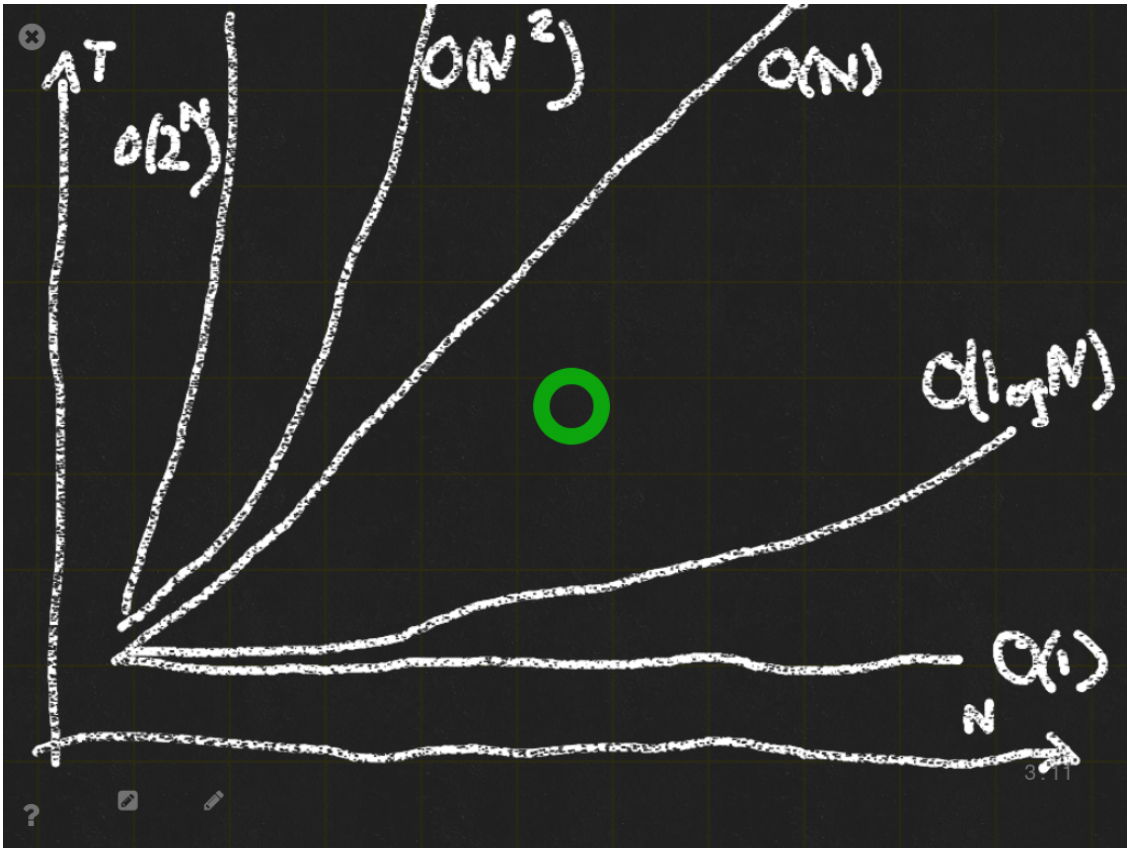
### 3.1 Come si comportano somme e prodotti degli ordini di grandezza

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

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

| la formula | diventa | perchè |
|---|---|---|
| costante * O(F(N)) | O(F(N)) | ignoriamo i fattori costanti |
| O(F(N)) + O(G(N)) | O(max(F(N),G(N))) | vogliamo il caso peggiore |
| O(F(N)) * O(G(N)) | O(F(N)*G(N)) | si moltiplicano |

**Esempi:** - $O(5 * n) ==> O(n)$ (si ignorano i fattori costanti) - $O(log(n)) + O(1) ==> O(log(n))$ (il logaritmo cresce mentre una costante no) - $O(log(n)) + O(n) ==> O(n)$ ( il logaritmo cresce più lentamente di n) - $O(n) * O(n) ==> O(n^2)$ - $O(n) * O(log(n)) + O(n) ==> O(n * log(n))$ (n*log cresce più rapidamente di n)



# 4 Nuovo vincolo: vogliamo usare poca memoria

Non ci serve ricordare o ordinare **tutti i valori** di L!!!!

Basta ricordare **SOLO k valori** (i migliori finora)

### 4.0.1 k massimi N numeri estratti a caso

- definisco ed inizializzo una lista vuota per i k valori (tempo costante)
- estraggo/leggo i valori e per ciascuno (N volte)
  - **aggiorno la lista dei k migliori già visti** (tempo ???)
- ritorno la lista dei k valori finale (tempo costante)

### 4.0.2 Per aggiornare i k valori con un nuovo X

- se la lista di valori ha meno di k elementi
- `aggiungo il nuovo valore (tempo costante)`
- se X è <= del minimo dei k valori (tempo $O(k)$)
- `lo posso ignorare (sono tutti meglio)`
- altrimenti è maggiore del minimo
- `tolgo il valore più piccolo (tempo $O(k)$)`
- `aggiungo il nuovo valore (tempo costante)`

[8]:
```
import random
# per ottenere i k massimi di tantissimi valori
# estratti a caso
def k_massimi_da_seq_casuale(k, N, seed):
    # settare il generatore al valore iniziale in modo da ripetere la stessa␣
 ↪sequenza
    random.seed(seed)
    # definisco ed inizializzo una lista vuota
    # per i k valori
    massimi = []
    # genero i valori in input e per ciascuno
    for _ in range(N):
        X = random.randint(-1_000_000,+1_000_000)
        # aggiorno la lista dei k migliori già visti
        aggiorna_k_massimi(massimi, X, k)
    # ritorno la lista dei k valori finale
    return massimi
```

[9]:
```
# per assicurarci che k_massimi_da_seq_casuale si comporti bene
# inventiamo un aggiornamento di prova per fare un test di funzionamento
def aggiorna_k_massimi_dummy(massimi, X, k):
    'versione che ricorda gli ultimi k valori letti, anche se non massimi, per␣
 ↪fare test'
```

```
    massimi.append(X)
    if len(massimi) > k:
        massimi.pop(0)
```

[10]:
```
# Controlliamo che k_massimi_da_seq_casuale funzioni
aggiorna_k_massimi = aggiorna_k_massimi_dummy
# ci aspettiamo gli ultimi 10 valori casuali
print(k_massimi_da_seq_casuale(k=10, N=10_000, seed=0))
# rigeneriamo la stessa sequenza e stampiamo gli ultimi 10
random.seed(0)
sequenza = [ random.randint(-1000000,+1000000) for _ in range(10_000) ]
print(sequenza[-10:])
```

```
[680056, -257049, 253844, 514597, -777285, 122459, -178847, -426915, 317230,
-581442]
[680056, -257049, 253844, 514597, -777285, 122459, -178847, -426915, 317230,
-581442]
```

[11]:
```
# Realizziamo l'aggiornamento come si deve
# per aggiornare la lista dei k valori con un nuovo valore X
def aggiorna_k_massimi(L, X, k):
    # se la lista di valori ha meno di k elementi
    if len(L) < k:
        # aggiungo il nuovo valore
        L.append(X)
        return
    # se X è minore o uguale del minimo dei k valori
    minimo = min(L)
    if X <= minimo:
        # lo posso ignorare
        return
    # altrimenti è maggiore del minimo
        # tolgo il valore più piccolo
    L.remove(minimo)
        # aggiungo il nuovo valore
    L.append(X)
```

[12]:
```
# Proviamo a vedere se due implementazioni diverse danno lo stesso risultato
valori = []
random.seed(0)
for _ in range(20):
    valori.append(random.randint(-1000000, +1000000))
valori.sort(reverse=True)
print(valori[:4])

k_massimi_da_seq_casuale(4,20,0)
```

```
[925676, 904450, 869947, 866976]
```

[12]:
```
[866976, 925676, 869947, 904450]
```

[23]:
```
%timeit k_massimi_da_seq_casuale(3,    1_000_000, 0)
%timeit k_massimi_da_seq_casuale(30,   1_000_000, 0)
%timeit k_massimi_da_seq_casuale(300,  1_000_000, 0)
%timeit k_massimi_da_seq_casuale(3000, 1_000_000, 0)
None
```

```
504 ms ± 8.34 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
736 ms ± 5.02 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
2.83 s ± 18.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
23.9 s ± 35.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

### 4.1 Nuova idea: tenere la lista di k elementi ordinata

e se tenessi la lista di k valori ordinata???

- trovare il minimo è costante (ultimo elemento)
- eliminare il minimo è costante
- per mantenere la lista ordinata
  - aggiungere l'elemento (tempo costante)
  - riordinare la lista (tempo $O(k * log(k))$

[13]:
```
# Versione che tiene ordinata la lista di K elementi
def aggiorna_lista_k_massimi_ordinati(Lordinata, X, K):
    # se len(Lordinata) < K
    if len(Lordinata) < K:
        # aggiungo il valore
        Lordinata.append(X)
        # mantengo ordinata la lista (tempo O(K*log(K)))
        Lordinata.sort(reverse=True)
        return
    # se X <= minimo
    if Lordinata[-1] >= X:
        # posso ignorarlo, esco
        return
    # altrimenti se X è maggiore del minimo
        # lo aggiungo e tolgo il minimo
    Lordinata[-1] = X
    # mantengo ordinata la lista (tempo O(K*log(K)))
    Lordinata.sort(reverse=True)
```

[14]:
```
aggiorna_k_massimi = aggiorna_lista_k_massimi_ordinati

# Tempo totale: proporzionale a N*K*log(K)
valori = []
random.seed(0)
for _ in range(20):
    valori.append(random.randint(-1_000_000, +1_000_000))
```

```
valori.sort(reverse=True)
print(valori[:4])

k_massimi_da_seq_casuale(4,20,0)
```

```
[925676, 904450, 869947, 866976]
```

[14]:
```
[925676, 904450, 869947, 866976]
```

[15]:
```
%timeit k_massimi_da_seq_casuale(3,    1_000_000, 34)
%timeit k_massimi_da_seq_casuale(30,   1_000_000, 34)
%timeit k_massimi_da_seq_casuale(300,  1_000_000, 34)
%timeit k_massimi_da_seq_casuale(3000, 1_000_000, 34)
None
```

```
454 ms ± 2.47 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
459 ms ± 748 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
468 ms ± 1.26 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
648 ms ± 1.71 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

#### 4.1.1 NOTA: L'aggiornamento della lista ordinata può essere fatto più rapidamente se

- troviamo la posizione dove inserire X con una ricerca binaria (tempo *O(log(k))*)
- e poi un insert (tempo $O(k)$)

Ovvero tempo $O(k + log(k)) ==> O(k)$

Prima avevamo $O(k * log(k))$ !!!

Quindi la soluzione potrebbe scendere a - Tempo totale: proporzionale a $N * K$ (se $K << N$)

**Implementazione della ricerca binaria per chi vuole, a casa**

#### 4.1.2 Vediamo che vuol dire $O(k * log(k) * N)$

| $k$ | $log(k)$ | $N$ -> | 1 | 10 | 100 | 1_000 | **10_000** |
|---|---|---|---|---|---|---|---|
| 10 | 3.32 | | 33 | 330 | 3_300 | 33_000 | 330_000 |
| 100 | 6.64 | | 664 | 6640 | 66_400 | 664_000 | 6_640_00 |
| 1_000 | 9.97 | | 9_970 | 99_700 | 997_000 | 9_970_000 | 99_700_000 |
| **10_000** | 13.29 | | 132_900 | 1_329_000 | 13_290_000 | 132_900_000 | **1_329_000_000** |

#### 4.1.3 E invece che vuol dire $O(k * N)$

| $k$ | $N$ | 1 | 10 | 100 | 1_000 | **10_000** |
|---|---|---|---|---|---|---|
| 10 | | 10 | 100 | 1000 | 10_000 | 100_000 |
| 100 | | 100 | 1_000 | 10_000 | 100_000 | 1_000_000 |
| 1_000 | | 1_000 | 10_000 | 100_000 | 1_000_000 | 10_000_000 |

| $k$ | $N$ | 1 | 10 | 100 | 1_000 | **10_000** |
|---|---|---|---|---|---|---|
| **10_000** | | 10_000 | 100_000 | 1_000_000 | 10_000_000 | **100_000_000** |

**Wooclap.com F23L**