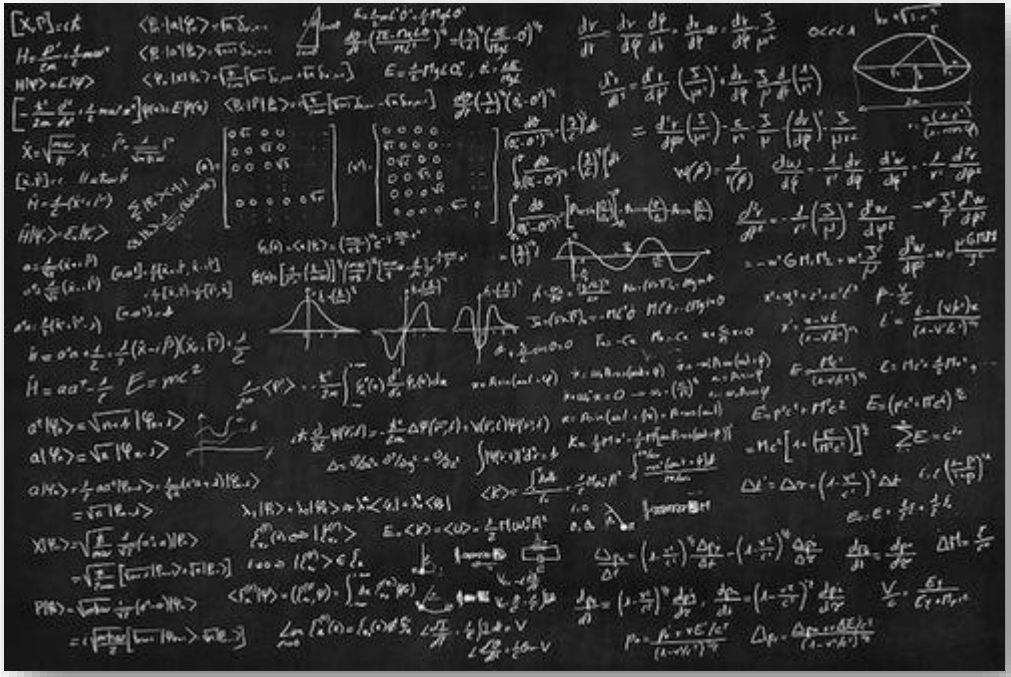


algoritmi notevoli

Python

- ***ricerca***
  (verificare la presenza di un valore in una sequenza)
  - ricerca ***sequenziale*** (sequenza non ordinata)
  - ricerca sequenziale (sequenza ordinata)
  - ricerca ***binaria*** (sequenza ordinata)
- ***ordinamento***
  (ordinare i valori all'interno di una sequenza in modo crescente o decrescente)
  - ***stupid*** Sort
  - ***selection*** Sort
  - ***bubble*** Sort
- ***merge***
  (fondere due sequenze ordinate in una terza sequenza)

- in generale un algoritmo di ricerca si pone come obiettivo quello di ***trovare*** un ***elemento*** avente determinate caratteristiche all'interno di un ***insieme*** di elementi
- nel nostro caso definiremo algoritmi che verificano la presenza di un valore in una sequenza (lista Python)
- la sequenza può essere non ordinata o ordinata
  - ipotizziamo l'ordinamento crescente

ricerca sequenziale

- la ***ricerca sequenziale*** (o completa) consiste nella ***scansione*** sequenziale degli elementi dal primo all'ultimo
- si interrompe quando il valore cercato è stato trovato, oppure quando si è sicuri che il valore non può essere presente
- ha il vantaggio di poter essere applicata anche a dati non ordinati
- negli esempi tratteremo la ricerca del valore **x** in una sequenza di elementi float di nome **v** con funzioni che restituiscono l'**indice** (la posizione) dell'elemento nella sequenza con valore x o **-1** in caso di valore non trovato

ricerca sequenziale


```
def ricerca(x: float, v: list) -> int:
    '''
    ricerca del valore x nella lista v
    restituisce posizione dell'elemento con valore x
    -1 se non presente
    '''
    for i in range(len(v)):
        if v[i] == x:
            return i
    return -1
```

ricerca sequenziale in sequenza ordinata

- se gli elementi sono ordinati …
  - supponiamo in ordine non decrescente
  - algoritmo speculare per ordinamento non crescente
- la ricerca sequenziale inizia sempre dal primo elemento
- si arresta quando l'elemento è stato trovato
- o quando non sono presenti più elementi
- o è stato analizzato un elemento con valore maggiore dell'elemento da ricercare
  - sicuramente l'elemento non sarà presente …

ricerca sequenziale in lista ordinata

A. Ferrari

```
def ricerca(x: float, v: list) -> int:
    '''
    ricerca del valore x nella lista ordinata v
    restituisce posizione dell'elemento con valore x
    -1 se non presente
    '''
    for i in range(len(v)):
        if v[i] == x:
            return i
        elif v[i] > x:
            return -1
    return -1
```

ricerca binaria o logaritmica

- l'algoritmo è simile al metodo usato per trovare una parola sul dizionario
  - sapendo che il vocabolario è ordinato alfabeticamente, l'idea è quella di iniziare la ricerca dall'elemento centrale (a metà del dizionario)
- il valore ricercato viene confrontato con il valore dell'elemento preso in esame:
  - se ***corrisponde***, la ricerca termina (l'elemento è stato trovato)
  - se è ***inferiore***, la ricerca viene ripetuta sugli elementi precedenti (sulla prima metà del dizionario), scartando quelli successivi
  - se invece è ***superiore***, la ricerca viene ripetuta sugli elementi successivi (sulla seconda metà del dizionario), scartando quelli precedenti
- se ***tutti*** gli elementi sono stati ***scartati***, la ricerca termina indicando che il valore non è stato trovato

ricerca binaria


```
def ricerca(x: float, v: list) -> int:
    '''
    ricerca del valore x nella lista ordinata v
    restituisce posizione dell'elemento con valore x
    -1 se non presente
    '''
    primo = 0
    ultimo = len(v)-1
    while primo < ultimo:
        medio = (primo + ultimo) // 2        #valore centrale
        if v[medio] == x:
            return medio; # x trovato alla posizione medio
        if v[medio] < x:
            primo = medio+1     # scarto la prima metà
        else:
            ultimo = medio-1    # scarto la seconda metà
    return -1                    # elemento non trovato
```

ricerca binaria ricorsiva

- l'algoritmo si presta ad una definizione ***ricorsiva***
- ad ogni chiamata della funzione si verifica se l'elemento ricercato si ***trova*** al centro dell'intervallo e in tal caso la funzione termina con ***successo***, in caso contrario si ***modifica*** l'***intervallo*** di ricerca e si effettua una nuova chiamata della funzione
- nel caso in cui l'intervallo di ricerca sia nullo si termina la ricorsione con insuccesso

```
def ricerca(x: float, v: list, primo: int, ultimo: int) -> int:
    '''
    ricerca del valore x nella lista ordinata v
    restituisce posizione dell'elemento con valore x
    -1 se non presente
    '''
    if primo > ultimo:
        return -1
    medio = (primo + ultimo) // 2        #valore centrale
    if v[medio] == x:
        return medio; # valore x trovato alla posizione medio
    if v[medio] < x:
        return ricerca(x,v,medio+1,ultimo)#scarto la prima metà
    else:
        return ricerca(x,v,primo,medio-1)#scarto la seconda metà
```

confronto fra gli algoritmi

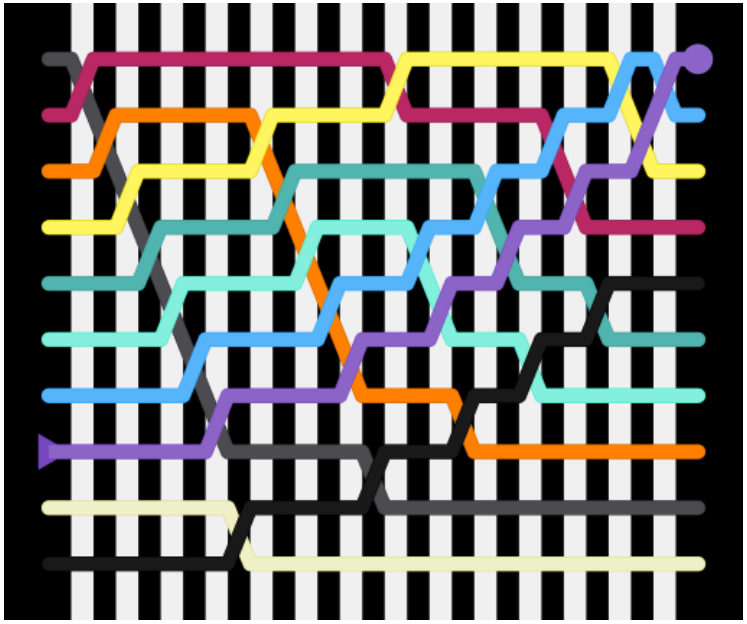
- l'efficienza si misura in base al numero di confronti effettuati che dipende da n (lunghezza della sequenza)
- si individuano il caso migliore e peggiore ma in generale interessa il caso medio

Algoritmo	Caso migliore	Caso peggiore	Caso medio	Caso medio con n = 1000
Ricerca Sequenziale	1	n	n / 2	500
Ricerca Binaria	1	lg_2 n	lg_2 n	10

Python: in

- la parola chiave in permette di verificare se un elemento è presente in una sequenza

```
valori = [2,5,1,12,8]
val = int(input('ricerca elemento: '))
if val in valori:
    print('valore presente')
else:
    print('valore non presente')
```



algoritmi di ordinamento

- ***stupid sort*** ☺
  - particolarmente inefficiente, come si può intuire dal nome consiste nel mischiare in qualche modo gli elementi dell'array poi controllare se è ordinato e, se non lo è, ricominciare da capo
- ***selection sort***
  - consiste in più scansioni della sequenza: al termine della prima il primo elemento conterrà il valore minore, poi si proseguirà ordinando la parte successiva della sequenza
- ***bubble sort***
  - consiste nella scansione della sequenza elemento per elemento, scambiando i valori dei due elementi consecutivi, quando il primo è maggiore del secondo

stupid sort

- l'algoritmo è ***probabilistico***
- la ragione per cui l'algoritmo arriva quasi sicuramente a una conclusione è spiegato dal teorema della ***scimmia instancabile***: *ad ogni tentativo c'è una probabilità di ottenere l'ordinamento giusto, quindi dato un numero illimitato di tentativi, infine dovrebbe avere successo*
- il ***bozo sort*** è una variante ancora meno efficiente
  - consiste nel controllare se la sequenza è ordinata e, se non lo è, prendere due elementi casualmente e scambiarli (indipendentemente dal fatto che lo scambio aiuti l'ordinamento o meno)

stupid sort

A. Ferrari

```
def stupidSort(v: list):
    '''
    ordina la lista v
    con algoritmo stupid sort
    '''
    while not ordinata(v):
        shuffle(v)

def ordinata(v: list) -> bool:
    '''
    true se la lista v
    è ordinata in modo non decrescente
    '''
    for i in range(len(v)-1):
        if v[i] > v[i+1]:
            return False
    return True
```

bozo sort

A. Ferrari

```
def bozoSort(v: list) -> int:
    '''ordina la lista v
    con algoritmo bozo sort
    '''
    while not ordinata(v):
        first = randint(0,len(v)-1)
        second = randint(0,len(v)-1)
        v[first],v[second] = v[second],v[first]
    return v

def ordinata(v: list) -> bool:
    ''' true se la lista v è ordinata in modo non decrescente
    '''
    for i in range(len(v)-1):
        if v[i] > v[i+1]:
            return False
    return True
```

selection sort

- l'algoritmo ricerca l'elemento ***minore*** della regione della sequenza da ordinare e lo sposta all'***inizio*** della regione stessa
- ad ogni scansione viene spostato un elemento della sequenza nella posizione corretta
- l'ordinamento ha termine quando la regione considerata è costituita da un solo elemento

esempio

sequenza di partenza	1	23	4	-56	65	21	32	15	0	-3
Scansione 1	-56	23	4	1	65	21	32	15	0	-3
Scansione 2	-56	-3	4	1	65	21	32	15	0	23
Scansione 3	-56	-3	0	1	65	21	32	15	4	23
Scansione 4	-56	-3	0	1	65	21	32	15	4	23
Scansione 5	-56	-3	0	1	4	21	32	15	65	23
Scansione 6	-56	-3	0	1	4	15	32	21	65	23
Scansione 7	-56	-3	0	1	4	15	21	32	65	23
Scansione 8	-56	-3	0	1	4	15	21	23	65	32
Scansione 9	-56	-3	0	1	4	15	21	23	32	65

selection sort (1)


```
def selectionSort(v: list):
    '''
    ordina la lista v
    con algoritmo selection sort
    '''
    for s in range(len(v)-1):
        for i in range(s+1,len(v)):
            if v[i] < v[s]:
                v[i],v[s] = v[s],v[i]
```

selection sort (2)

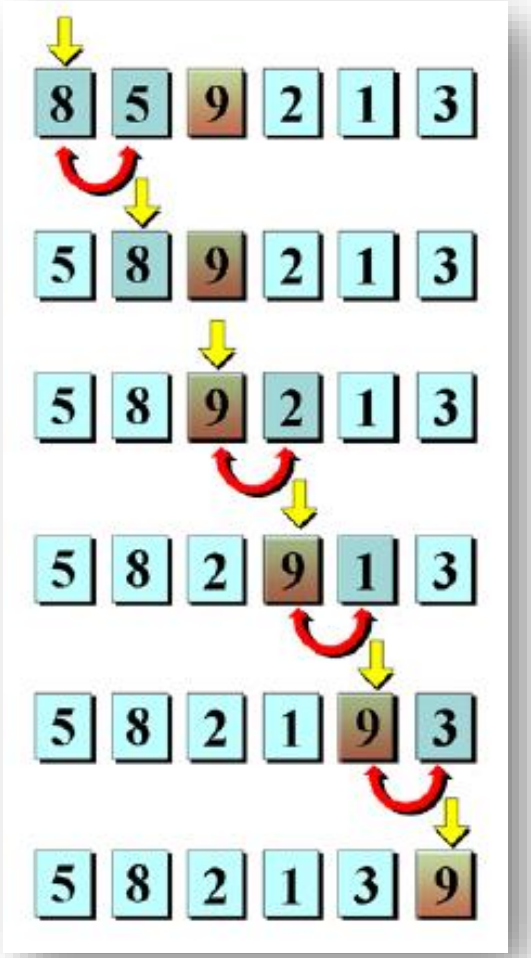

```
def selectionSort(v: list):
    '''
    ordina la lista v
    con algoritmo selection sort
    '''
    for s in range(len(v)-1):
        posMin = s
        for i in range(s+1,len(v)):
            if v[i] < v[posMin]:
                posMin = i
        v[posMin],v[s] = v[s],v[posMin]
```

bubble sort

- consiste nella scansione dell'array elemento per elemento, scambiando i valori dei due ***elementi consecutivi***, quando il primo è maggiore del secondo
- al termine della scansione, in genere l'array non risulta ordinato e si deve procedere a una nuova scansione e alla conseguente serie di eventuali scambi tra i valori di due elementi consecutivi
- sicuramente l'array risulta ordinato quando si sono effettuate **`n - 1`** scansioni, se n sono gli elementi dell'array
- è detto *bubblesort* (ordinamento a bolle) per analogia con le bolle d'aria nell'acqua che, essendo leggere, tendono a spostarsi verso l'alto

bubble sort - esempio

A. Ferrari



bubble sort


```
def bubbleSort(v: list):
    '''
    ordina la lista v
    con algoritmo bubble sort
    '''
    for s in range(len(v)):
        for i in range(len(v)-s-1):
            if v[i] > v[i+1]:
                v[i],v[i+1] = v[i+1],v[i]
```

bubble sort (2)

è possibile ***migliorare*** l'efficienza dell'algoritmo controllando se sono stati effettuati ***spostamenti***, in caso negativo l'array risulta già ***ordinato***

```
def bubbleSort(v: list):
    ''' ordina la lista v
    con algoritmo bubble sort '''
    spostamento = True
    s = 0
    while spostamento and s < len(v) -1:
        spostamento = False
        for i in range(len(v)-s-1):
            if v[i] > v[i+1]:
                v[i],v[i+1] = v[i+1],v[i]
                spostamento = True
        s += 1
```

sort Python

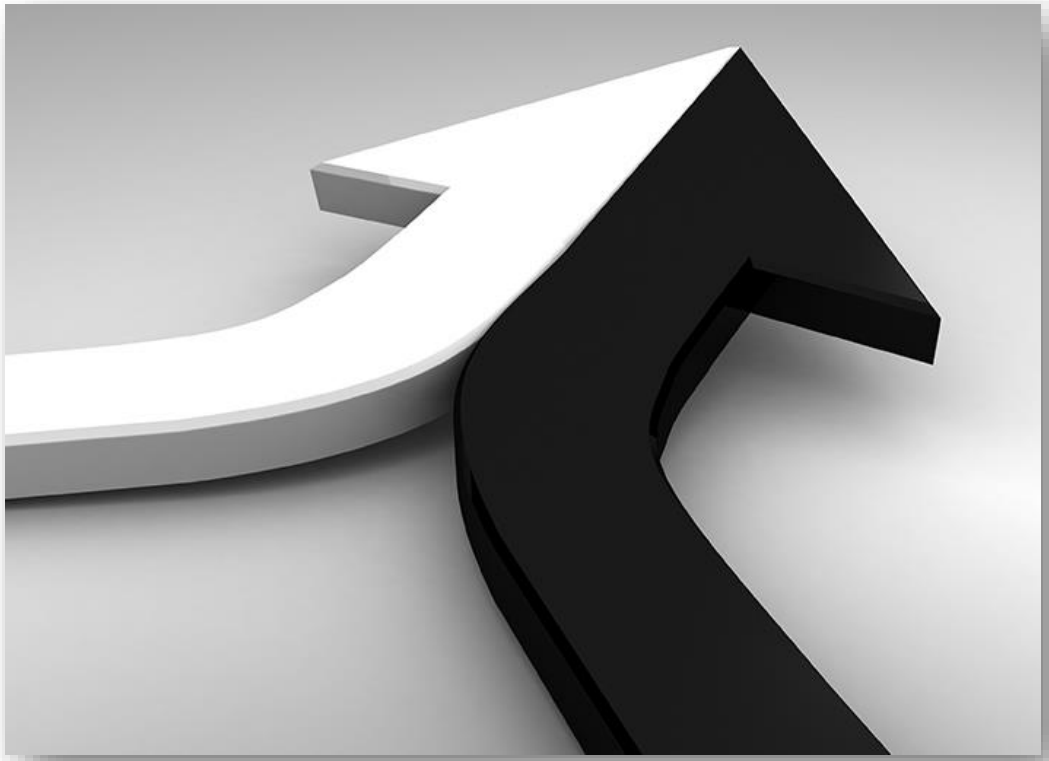
A. Ferrari

- Python mette a disposizione il metodo sort per ordinare sequenze
- è un algoritmo efficiente [complessità O(n log n)]
- l'algoritmo usato è il Timsort derivato dal merge sort e dall'insertion sort

```
valori = [5, 4, 6, 12, 8]
print(valori)
valori.sort()
print(valori)
```

confronto

https://www.toptal.com/developers/sorting-algorithms

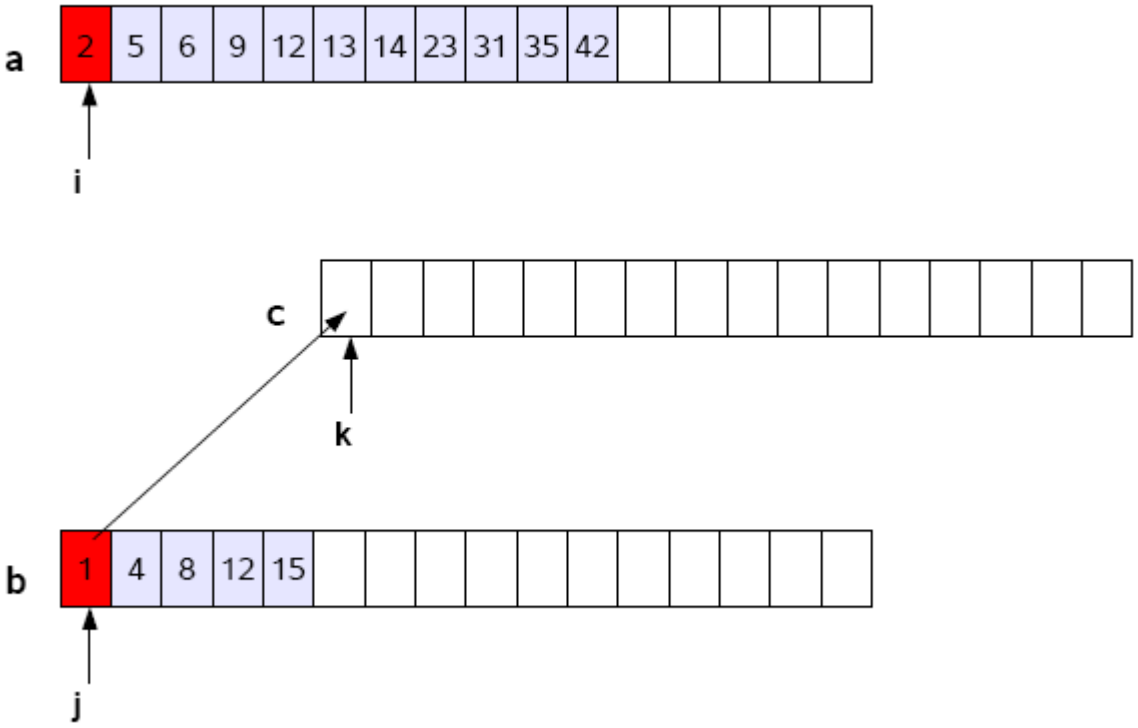


merge

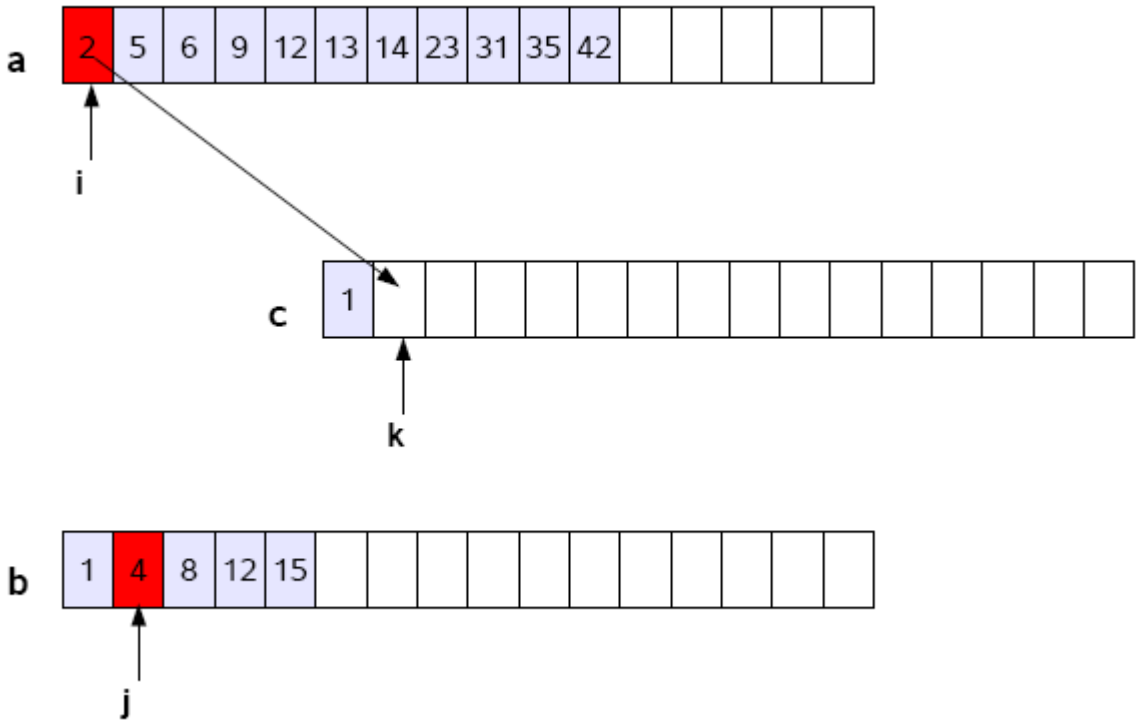
fusione di due sequenze ordinate

- si parte da due sequenze ordinate ***a*** e ***b*** per ottenere una terza sequenza ***c*** ordinata e contenente sia i dati presenti in ***a*** che quelli presenti in ***b***
- l'algoritmo prevede la scansione delle due sequenze con due indici diversi trasferendo in ***c*** l'elemento con valore minore

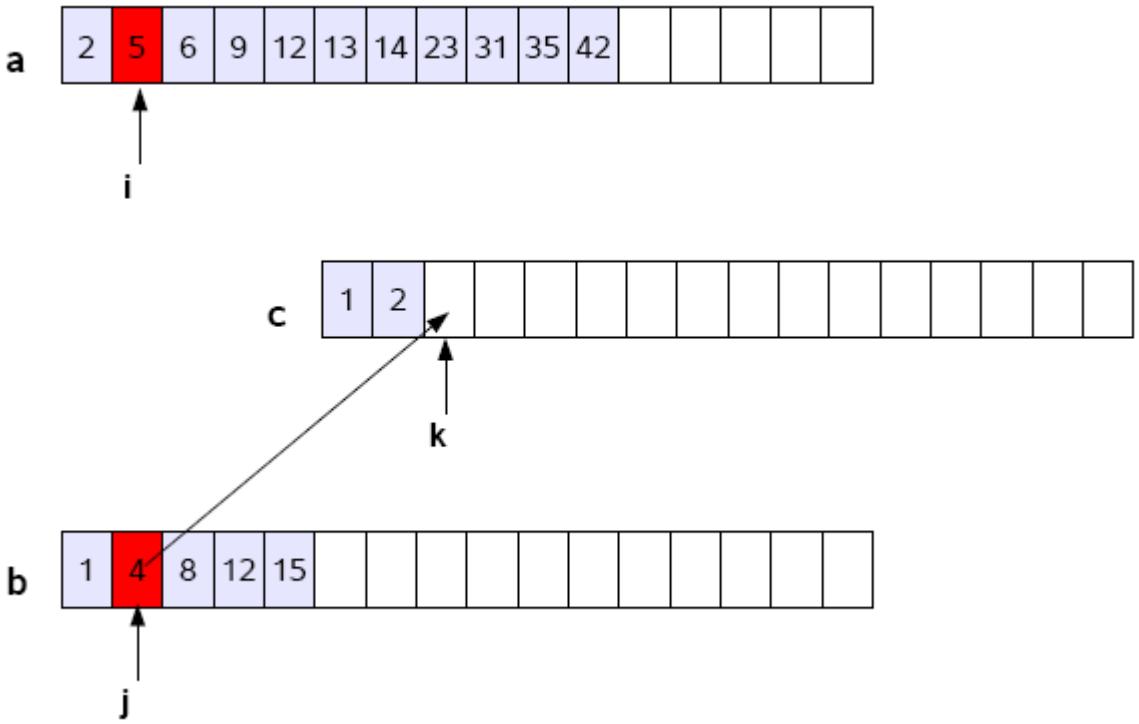
passaggio 1



passaggio 2



passaggio 3



```
def merge(a: list, b: list) -> list:
    '''
    restituisce la lista ordinata ottenuta
    dalla fusione delle liste ordinate a e b
    '''
    ia = ib = 0
    v = []
    while ia < len(a) and ib < len(b):
        if a[ia] < b[ib]:
            v.append(a[ia])
            ia += 1
        else:
            v.append(b[ib])
            ib += 1
    while ia < len(a):
        v.append(a[ia])
        ia += 1
    while ib < len(b):
        v.append(b[ib])
        ib += 1
    return v
```