

# *Ottimizzazioni di mergeSort*

corso di laurea in **Matematica**

*Informatica Generale*, Lezione **9.3**

**Ivano Salvo**



**SAPIENZA**  
UNIVERSITÀ DI ROMA

# Funzione fondi e ottimizzazioni

Ecco la funzione *fondi*.

Purtroppo abbiamo bisogno di un **vettore d'appoggio**  $z$ : questo perché non possiamo prevedere quanti elementi arrivano dal primo vettore e quanti dal secondo: *merge* **non si può fare in place**.

Abbiamo scritto una funzione *merge* molto generale, che qui viene usata in modo particolare: **fonde due parti dello stesso vettore**.

Si può **dimezzare il numero delle copiatore**, con un'astuzia: si copia la prima metà di  $v$  in  $z$  e poi fondiamo  $v$  (seconda metà) e  $z$  in  $v$ .

```
def fondi(v, inf, m, sup):  
    z = alloca(sup - inf)  
    merge(v, inf, m, v, m, sup, z, 0)  
    copia(z, 0, sup-inf, v, inf)
```

```
def fondi(v, inf, m, sup):  
    z = alloca(m - inf)  
    copia(v, inf, m, z, 0)  
    merge(z, 0, m-inf, v, m, sup, v, inf)
```

# Esempio di esecuzione di mergesort

Vediamo un esempio di traccia: osservate come mergeSort va prima a sinistra, poi a destra.

Osservate anche che quando fondo vettori più grandi, non avrei più bisogno dei vettori d'appoggio più piccoli: posso usarne **1 solo!**

	lo	hi	a[]													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
merge(a, 0, 0, 1)																
merge(a, 2, 2, 3)																
merge(a, 0, 1, 3)																
merge(a, 4, 4, 5)																
merge(a, 6, 6, 7)																
merge(a, 4, 5, 7)																
merge(a, 0, 3, 7)																
merge(a, 8, 8, 9)																
merge(a, 10, 10, 11)																
merge(a, 8, 9, 11)																
merge(a, 12, 12, 13)																
merge(a, 14, 14, 15)																
merge(a, 12, 13, 15)																
merge(a, 8, 11, 15)																
merge(a, 0, 7, 15)																

Trace of merge results for top-down mergesort

# Ottimizzazioni: un solo vettore *z*

È quindi sufficiente un unico vettore di appoggio.

Occorre quindi **passare** sui parametri un **riferimento** a ***z*** (**allocato una sola volta** alla chiamata iniziale).

Occorre modificare *fondi* coerentemente.

Notare che usiamo *z* sulle stesse porzioni di *v* che sto ordinando.

*z* si usa **nella stessa porzione**  $[inf, sup]$  di *v*

```
def mergeSort(v):  
    z = alloca(len(v))  
    mergeSortAux(v, 0, len(v), z)  
  
def mergeSortAux(v, inf, sup, z):  
    if sup - inf < 2:  
        m = puntoMedio(inf, sup)  
        mergeSortAux(v, inf, m, z)  
        mergeSortNaif(v, m, sup, z)  
        fondi(v, inf, m, sup, z)
```

```
def fondi(v, inf, m, sup, z):  
    copia(v, inf, m, z, inf)  
    merge(z, inf, m, v, m, sup, v, inf)
```

# Quello che gli algoritmisti non dicono

Idea: **alternare il ruolo di  $v$  e  $z$**  durante le discese ricorsive: questo **evita le copie**, al prezzo di spostare il puntatore al vettore e non sapere se il risultato sia in  $z$  o in  $v$ : passiamo il risultato!

```
def mergeSort(v):
    z = alloca(len(v))
    # torna un riferimento al vettore
    # ordinato, che può essere z o v
    v, z = mergeSortAux(v, z, 0, len(v))
    return v # il primo è il risultato

def mergeSortAux(v, inf, sup, z):
    if sup - inf < 2:
        m = puntoMedio(inf, sup)
        v, z = mergeSortAux(v, inf, m, z)
        v, z = mergeSortNaif(v, m, sup, z)
        fondi(v, inf, m, sup, z)
    else: z[inf] = v[inf]
    return z, v # inverte z e v!
```

*devo comunque  
invertire z e v, per  
essere coerente con  
le chiamate sorelle*

# *mergeSort iterativo*

A ben vedere, le suddivisioni di mergeSort sono **sempre le stesse, indipendentemente dai valori** contenuti nel vettore.

È facile scrivere quindi la versione iterativa **bottom-up**.

Sappiamo  $\text{Asc}(v[i, i+1])$  è sempre vero e sappiamo che avendo  $\text{Asc}(v[i, i+l])$  e  $\text{Asc}(v[i+l, i+2\cdot l])$  possiamo facilmente costruire una porzione di vettore che soddisfa  $\text{Asc}(v[i, i+2\cdot l])$  invocando *fondi*.

Assumiamo per semplicità  $n=2^k$ , per qualche  $k$ . L'estensione al caso generale è semplice, ma **noiosa** (verificare cosa accade nello sfrido)

```
def mergeSortBottomUp(v):
    z, l = alloca(len(v)), 1
    while l < n:
        # INV:  $\forall k \in [0, n/l). \text{Asc}(v[kl, (k+1)*l])$ 
        i=0 # inizio seq. da fondere
        while i < n:
            fondi(v, i, i+l, i+2*l, z)
            i = i+2*l #inizio seq. successiva
        l = l*2 # passo a sequenze più lunghe
```

*il ciclo interno modifica  $v$  che soddisfa  $\varphi(v, l)$  in  $v'$  che soddisfa  $\varphi(v', 2l)$*

# Esecuzione di mergeSort bottom-up

	a[i]															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<b>sz = 2</b>	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 4, 5)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 6, 6, 7)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 8, 8, 9)	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
merge(a, 10, 10, 11)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, 12, 12, 13)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, 14, 14, 15)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L
<b>sz = 4</b>	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L
merge(a, 0, 1, 3)	E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L
merge(a, 4, 5, 7)	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
merge(a, 8, 9, 11)	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
merge(a, 12, 13, 15)	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P
<b>sz = 8</b>	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, 0, 3, 7)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge(a, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
<b>sz = 16</b>	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X
merge(a, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Trace of merge results for bottom-up mergesort

# Problema: punti coincidenti

[Esame del 27/1/2022]

**Problema:** Dati due vettori,  $u[0, m)$  e  $v[0, n)$ , contenenti ciascuno valori distinti (cioè  $i \neq j \Rightarrow u[i] \neq u[j]$  e  $v[i] \neq v[j]$ ) determinare il numero di elementi comuni, formalmente:

$$\text{PC}(u, v) = \# \{(i, j) : 0 \leq i < m, 0 \leq j < n \text{ e } u[i] = v[j]\}$$

- a) scrivere una funzione **puntiCoincidenti(u, v)** che risolve il problema per ogni vettore  $u$  e  $v$ .
- b) scrivere una funzione **puntiCoincidentiOrd(u, v)** nel caso in cui  $u$  e  $v$  siano **ordinati** in modo crescente.
- c) dovendo risolvere il caso generale e potendo modificare i vettori, conviene usare la funzione `puntiCoincidenti` oppure prima ordinare i vettori  $u$  e  $v$  e poi chiamare `puntiCoincidentiOrd`?

# Punti coincidenti: caso generale (1)

Una soluzione a **forza bruta**, consiste nell'esaminare **tutte le coppie** di indici  $(i, j)$  e contare quante volte  $u[i] = v[j]$ .

Questo può essere fatto con **due cicli annidati** e un **contatore**.

**Analisi:**

- il **confronto** e l'**incremento** di  $c$  hanno tempo **costante**  $\theta(1)$
- il **ciclo interno** fa sempre  $n$  **iterazioni**, quindi è  $\theta(n)$ .
- il **ciclo esterno** ripete sempre questa operazione **sempre**  $m$  **volte**.

Essendo **cicli annidati**, le complessità si **moltiplicano**.

La complessità totale è quindi  $\theta(m \cdot n)$ .

```
def puntiCoincidenti(u, v):  
    m, n, c = len(u), len(v), 0  
    for i=0 to m-1:  
        for j=0 to n-1:  
            if u[i]==v[j]:  
                c=c+1  
    return c
```

$\theta(m \cdot n)$

$\theta(m)$

$\theta(n)$

$\theta(1)$

**I furbi invece...**

```
def puntiCoincidenti(u, v):  
    m, n, c = len(u), len(v), 0  
    for i=0 to m-1:  
        c = c + conta(u[i], v)  
    return c
```

$\theta(m \cdot n)$

$\theta(n)$

## Punti coincidenti: caso generale (2)

Il programma visto ha come pregi la **semplicità**, e **funziona anche oltre le precondizioni**, quando i due vettori **contengono ripetizioni**.

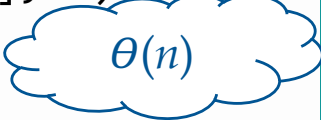
Possiamo però usare le precondizioni per **risparmiare qualcosa**?

**Sostituendo** il ciclo interno con la **ricerca sequenziale**, il ciclo interno si arresta appena trovo  $u[i]$  in  $v$ .

Risparmiamo lavoro, **ma non si modifica la complessità asintotica** di caso pessimo, che rimane  $\theta(m \cdot n)$  visto che la ricerca è  $\theta(n)$ .

Osservate l'**invariante**:  $c = \text{PC}(u[0, i], v)$  e  $\text{PC}(u[0, m], v) \equiv \text{PC}(u, v)$

```
def puntiCoincidenti (u, v):  
    # REQ: u, v di elementi distinti  
    m, c = len(u), 0  
    for i=0 to m-1:  
        # INV: c = PC(u[0, i], v)  
        if ricerca(u[i], v) > -1:  
            c=c+1  
    return c
```



Si migliora il in media,  
perché si fanno  $n/2$  confronti  
...ma  $\theta(m \cdot n / 2) = \theta(m \cdot n)$

# *l'ordine migliora la vita...*

Viceversa, il programma che usa la ricerca sequenziale, può **ispirare** un immediato ed efficace miglioramento di complessità nel caso ordinato (in questo caso è **sufficiente  $v$  ordinato**).

Infatti, sotto la precondizione che  $v$  sia ordinato posso chiamare la funzione **ricercaBinaria** in quanto **rispetto le sue precondizioni**.

**Conseguenza:** lo stesso programma, cambiando la funzione chiamata, diventa di **complessità  $\theta(m \log n)$**  in quanto esegue  $m$  ricerche binarie nel vettore  $v$  (costo  $\log n$ ).

```
def puntiCoincidenti(u, v):  
    # REQ: Asc(v)  
    m, c = len(u), 0  
    for i=0 to m-1:  
        # INV: c = PC(u[0,i], v))  
        if ricercaBinaria(u[i], v) > -1:  
            c=c+1  
    return c
```

$\theta(m \cdot \log n)$

$\theta(\log n)$

# Esploriamo altre strade...

La funzione  $\mathbf{PC}(u, v)$  è additiva su segmenti di vettori. Cioè:

$$\forall k \in [0, m). \mathbf{PC}(u[0, n], v) = \mathbf{PC}(u[0, k], v) + \mathbf{PC}(u[k, m), v)$$

$$\forall k \in [0, n). \mathbf{PC}(u, v[0, n]) = \mathbf{PC}(u, v[0, k]) + \mathbf{PC}(u, v[k, n])$$

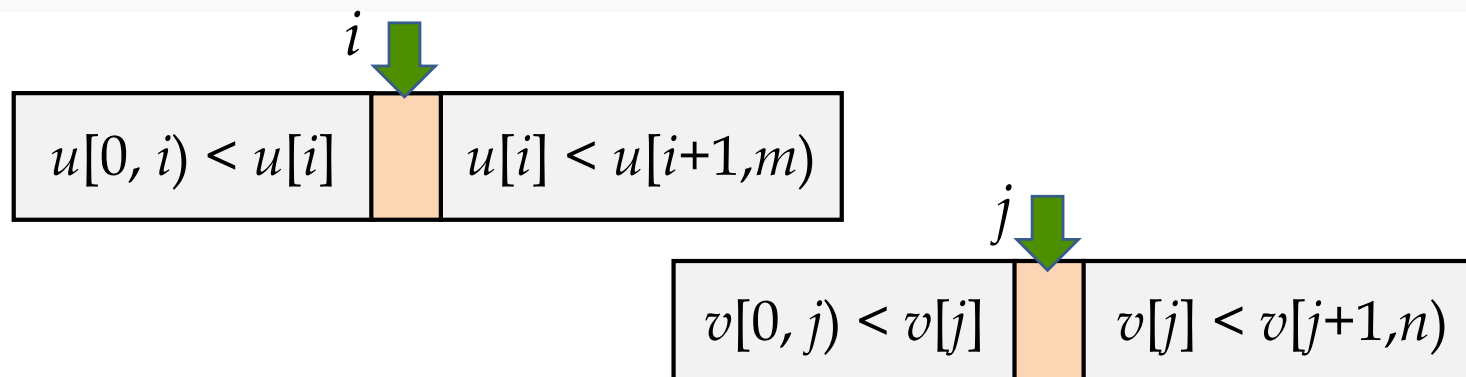
Questa proprietà è necessaria per far vedere che l'invariante si conserva nei programmi precedenti, in quanto:

$$\mathbf{PC}(u[0, i+1), v) = \mathbf{PC}(u[0, i), v) + \mathbf{PC}(u[i], v)$$

Applicando questa proprietà, presi due indici  $i \in [0, m), j \in [0, n)$

$$\begin{aligned} \mathbf{PC}(u[0, m), v[0, n]) &= \mathbf{PC}(u[0, i), v[0, j]) + \mathbf{PC}(u[i, m), v[0, j]) + \\ &+ \mathbf{PC}(u[0, i), v[j, n]) + \mathbf{PC}(u[i, m), v[j, n]) \end{aligned}$$

Cosa possiamo dedurre da  $\mathbf{Cr}(u), \mathbf{Cr}(v)$  confrontando  $u[i]$  e  $v[j]$ ?



## ... qualche buona notizia ...

Se  $u[i] = v[j]$ , da  $\mathbf{Cr}(u)$  &  $\mathbf{Cr}(v)$  e transitività, possiamo dedurre:

$$u[0, i) < u[i] = v[j] < v[j+1, m) \quad \text{e} \quad u[i+1, n) > u[i] \geq v[j] > v[0, j)$$

Questo implica anche  $\mathbf{PC}(u[0, i), v[j, n)) = 0$  e  $\mathbf{PC}(u[i, m), v[0, j)) = 0$  che unito alla formula generale, implica in particolare:

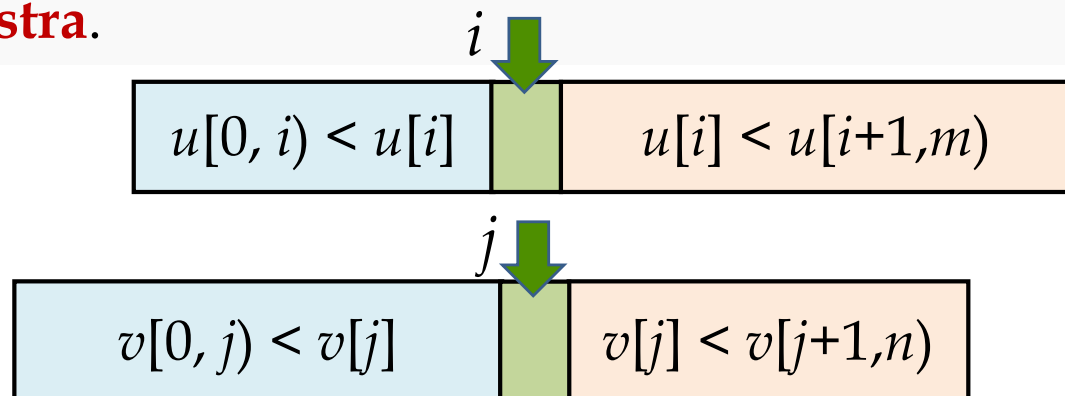
$$\mathbf{PC}(u[0, m), v[0, n)) = 1 + \mathbf{PC}(u[0, i), v[0, j)) + \mathbf{PC}(u[i+1, m), v[j+1, n))$$

cioè, posso controllare **separatamente le metà sinistre e destre**.

Sapendo che dobbiamo progettare una scansione iterativa dei due vettori, da sinistra a destra, possiamo pensare di avere una variabile  $c$  per cui valga la proprietà **invariante**  $c = \mathbf{PC}(u[0, i), v[0, j))$ . Da cui:

$$c = \mathbf{PC}(u[0, i), v[0, j)) \ \& \ u[i]=v[j] \Rightarrow c + 1 = \mathbf{PC}(u[0, i+1), v[0, j+1))$$

Questo **invariante** si conserva incrementando  $c$ ,  $i$  e  $j$  e posso continuare **andando a destra**.



## ... e qualche difficoltà da risolvere...

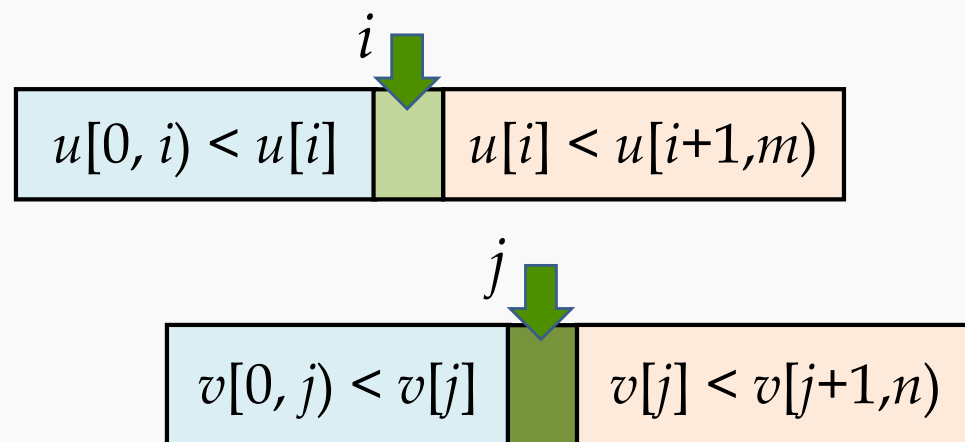
Se  $u[i] < v[j]$ , posso analogamente dedurre

$$u[0, i] < u[i] < v[j] \leq v[j, m) \text{ e quindi } \mathbf{PC}(u[0, i), v[j, n)) = 0,$$

ma **non posso dedurre**  $\mathbf{PC}(u[i, m), v[0, j)) = 0$ : questo ha la spiacevole conseguenza che dovrei continuare a contare sia a destra che a sinistra.

Come si vede in figura, immaginando che  $u[i]$  sia in corrispondenza del  $v[j]$  a lui più vicino, non so niente di  $\mathbf{PC}(u[i, m), v[0, j))$

Problema simmetrico nel caso  $u[i] > v[j]$  con  $\mathbf{PC}(u[0, i), v[j+1, n))$ .



## ... soluzione delle difficoltà ...

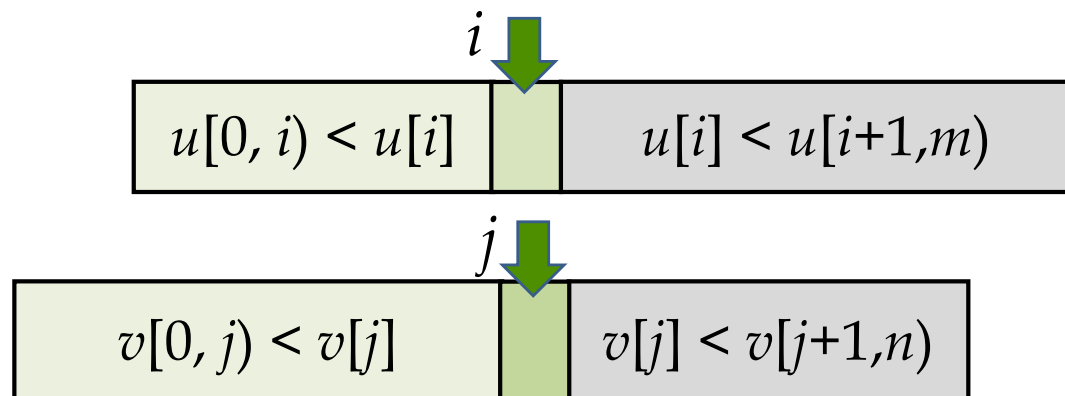
Tuttavia, se sapessimo per qualche motivo che  $u[i] > v[0, j)$  avremo ovviamente  $\text{PC}(u[i, m), v[0, j)) = 0$  e quindi il **via libera verso destra** e **simmetricamente**  $v[j] > u[0, i)$  ci permetterebbe di gestire il caso simmetrico  $u[i] > v[j]$ .

Possiamo **rinforzare l'invariante** con queste proprietà?

Ricordiamo che  $u[i] > v[0, j)$  implica  $u[i+1] > v[0, j)$ , e nel caso  $u[i] < v[j]$  implica anche  $v[j] > u[0, i+1)$ .

Simmetricamente  $v[j] > u[0, i)$  implica  $v[j+1] > u[0, i)$ , e nel caso  $u[i] > v[j]$  implica anche  $u[i] > v[0, j+1)$ . Abbiamo quindi un candidato **invariante buono** per **tutti i casi**:

$$\varphi(c, i, j) \equiv \text{PC}(u[0, i), v[0, j)) = c \ \& \ u[i] > v[0, j) \ \& \ v[j] > u[0, i)$$



## ... gran finale

L'invariante appena discusso è **soddisfacibile** banalmente su **segmenti vuoti** di vettore, infatti:

$$\varphi(0, 0, 0) \equiv \mathbf{PC}(u[0, 0), v[0, 0)) = 0 \ \& \ u[0] > v[0, 0) \ \& \ v[0] > u[0, 0)$$

semplicemente perché segmenti  $v[0, 0)$  e  $u[0, 0)$  sono vuoti, quindi abbiamo l'inizializzazione  $c, i, j = 0, 0, 0$ .

La discussione precedente, ha invece dimostrato che:

$$\varphi(c, i, j) \ \& \ u[i]=v[j] \text{ implica } \varphi(c+1, i+1, j+1)$$

$$\varphi(c, i, j) \ \& \ u[i]<v[j] \text{ implica } \varphi(c, i+1, j)$$

$$\varphi(c, i, j) \ \& \ u[i]>v[j] \text{ implica } \varphi(c, i, j+1)$$

Quindi, in ogni caso, **incremento**  $i, j$  o **entrambi**. Questo implica che la funzione  $t(m, n, i, j) = m + n - i - j$  è una funzione di **terminazione** e dà anche il limite superiore  $\Theta(m+n)$  alle iterazioni.

Osserviamo infine che, finito uno dei due vettori, ovviamente abbiamo finito, perché il **conteggio sul segmento vuoto è sempre 0**.

## ... e adesso lo pseudocodice

Ecco lo pseudocodice. Guardate l'invariante in forma sintetica.

È uno **schema di algoritmo** molto **diffuso** in problemi tra vettori ordinati. Per es.: “**intersezione**” di elementi tra due vettori ordinati,

Vedremo un esempio notevole nel cuore di un importante algoritmo di ordinamento: **fusione ordinata di vettori ordinati**, aka **merge**.

```
def coincidenceCount(u,v):
    m, n = len(u), len(v)
    c, i, j = 0, 0, 0
    while i<m and j<n:
        #INV: c + PC(u[i,n), v(j,n)) = PC(u, v)
        if u[i]==v[j]:
            c, i, j = c+1, i+1, j+1
        else if u[i] < v[j]: i = i+1
        else j = j+1
    return c
```