# *Soluzioni 5bis L'Angolo Degli Esercizi*

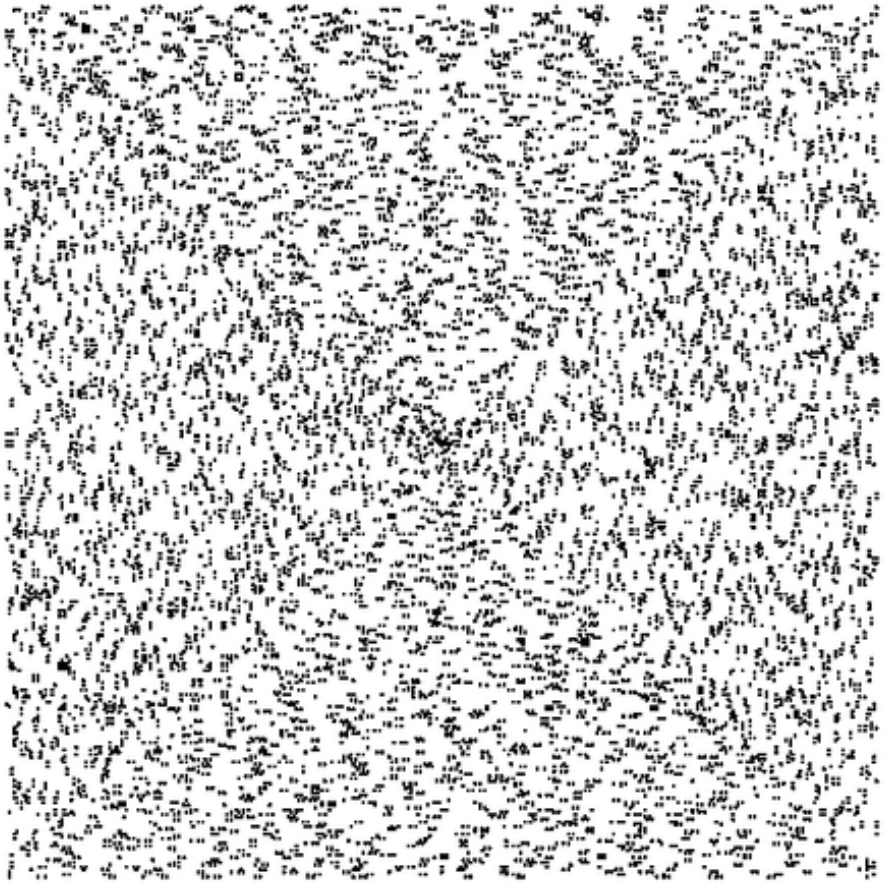
corso di laurea in **Matematica**

*Informatica Generale*, **Ivano Salvo**

Esercizi lez. **...** [-/-/--]


SAPIENZA
UNIVERSITÀ DI ROMA

# *Calcolo degli*
# *Ulam numbers*



(1971), 249–257. These mysterious numbers, which were first defined by S. Ulam in *SIAM Review* **6** (1964), 348, have baffled number theorists for many years. The ratio

# Back to Ulam numbers

Supponiamo di conoscere i primi $n$ numeri di Ulam memorizzati in un vettore ordinato $u = u_0 < u_1 < \ldots < u_{n-1}$. Vediamo come calcolare il prossimo.

Sappiamo che $u_n \in [u_{n-1}+1, u_{n-1}+u_{n-2}]$. Partendo da $x = u_{n-1}+1$ verifichiamo se $x$ è somma **di esattamente una coppia** di elementi in $u$. Se sì, x è $u_n$ altrimenti proviamo $x$+1 e così via.

Il programma risultante è **quadratico**: `contaSomme` ha complessità lineare in $n$, e **devo fare in media 13.5 tentativi** per trovare il prossimo Ulam number (questo valore è sperimentale ma è approssimativamente costante).

Rispetto a `contaSomme` ci interessa sapere solo se il numero di somme sia 0, 1 oppure >1 e quindi usciamo non appena troviamo la seconda somma uguale a $x$+1, senza modificare la complessità asintotica (**ma accelerando significativamente il programma**).

$U_n/n$ appears to converge to a constant, $\approx 13.52$; for example, $U_{20000000} = 270371127$

# Ulam: il codice del Professore

```
int nextU(int *u, int k){
   int c,s,inf,sup;
   int nu = u[k-1];
   do {
      c=0; nu++;
      inf=0; sup=k-1;
      while (inf<sup){
         s = u[inf]+u[sup];
         if (s<nu) inf++;
         else if (s>nu) sup--;
         else {
            if (c) {c=0; break;} else c=1;
            inf++; sup--;
         }
      } /* end while */
   } while (!c)
   return nu;
}
```

```
int ulam(int n){
   int u[n+1];
   u[0]=1; u[1]=2;
   for (int i=2; i<=n; i++)
      u[i]=nextU(u,i);
   return u[n];
}
```

# Ulam "generativo"

Abbiamo programmato un `**crivello di Ulam**': si provano tutti i naturali e si tengono solo quelli riconosciuti soddisfare i requisiti per essere il prossimo numero di Ulam. Un'alternativa è quella di **generare tutte le somme**.

Lavorando con array, è possibile tenere un array $s$ che riporta in posizione $i$ il numero di coppie di distinti numeri di Ulam che generano $i$ come somma.

**Attenzione:** generare 1 sola volta le somme. Trovato $u_n$ si possono incrementare tutti i valori $s_j$ tali che $j = u_i+u_n$ per $i<n$.

Il problema più serio è **quante posizioni allocare per $s$**: possiamo speculare sul fatto che $\lim_{n\to\infty} u_n/n$=13.52 e quindi allocare $27n$ posizioni (ricordiamo però che generemo le somme fino a $u_{n-1}+u_{n-2}$). **Con le liste questa idea è più elegante** [**Esercizio**]

constant, $\approx 21.6016$. Calculations by Jud McCranie and the author for $U_n < 640000000$ indicate that the largest gap $U_n - U_{n-1}$ may occur between $U_{24576523} = 332250401$ and $U_{24576524} = 332251032$; the smallest gap $U_n - U_{n-1} = 1$ apparently occurs only when $U_n \in \{2, 3, 4, 48\}$. Certain small gaps like 6, 11, 14, and 16 have never been observed.]

# Ulam: un programma "generativo"

```
int ulam(int n){
   int u[n+1];
   u[0]=1; u[1]=2;

   int *s = (int*) calloc(27*n,sizeof(int));
   int nu=2;

   for (int i=1; i<n; i++){
          /* aggiorna le somme con u[i] */
      for (int j=i-1; j>=0; j--)
                    s[u[j]+u[i]]++;
          /* cerca il prox */
      while (s[nu]!=1) nu++;
          /* carica il nuovo Ulam number
           * sposta l'inizio nuova ricerca*/
      u[i+1]=nu++;
   }
   return u[n];
}
```

# *Esercizio 4.3*
# *Punti Coincidenti*
# *(con tutti le asserzioni)*

# *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:

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

**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`?

**d**) Modificare il programma che risolve il punto **b**) in modo che risolva in tempo lineare il caso con ripetizioni. Osservare che ogni algoritmo che conta gli elementi uguali uno ad uno ha tempo di esecuzione (pessimo) $\boldsymbol{\Omega}(nm)$ e non può quindi essere lineare.

# *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**: $c$ = **PC**$(u[0, i), v)$ e **PC**$(u[0, m), v)$ ≡ **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
```

$\theta(n)$

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 $\log n$).

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

```
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
```

$\theta(\log n)$

# *Se i vettori contengono valori in [0, k)*

In questo caso, è sufficiente:

**1**. allocare due vettori di lunghezza $k$, $cu$ e $cv$.

**2**. usarli per contare, per ogni $i \in [0, k)$ quanti elementi ho nei vettori u e v di valore i, formalmente $cu[i] = \#\{ j \mid u[j] = i \}$

**3**. a quel punto, il coincident count (anche avendo ripetizioni) è semplicemente $\mathbf{cc}(u, v) = \sum_{0 \le i < k} cu[i] \cdot cv[i]$.

Questo programma conta correttamente i punti coincidenti anche quando ci **sono ripetizioni**!

```
def puntiCoincidenti0K(u, v):
    cu = conta(u, k)
    cv = conta(v, k)
    pc = 0
    for i=0 to k-1:
        pc = pc + cu[i] * cv[i]
    return pc
```

$\Theta(n+k)$

$\Theta(n+k)$

$\Theta(k)$

# *Esploriamo altre strade...*

La funzione **PC**$(u, v)$ è additiva su segmenti di vettori. Cioè:

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

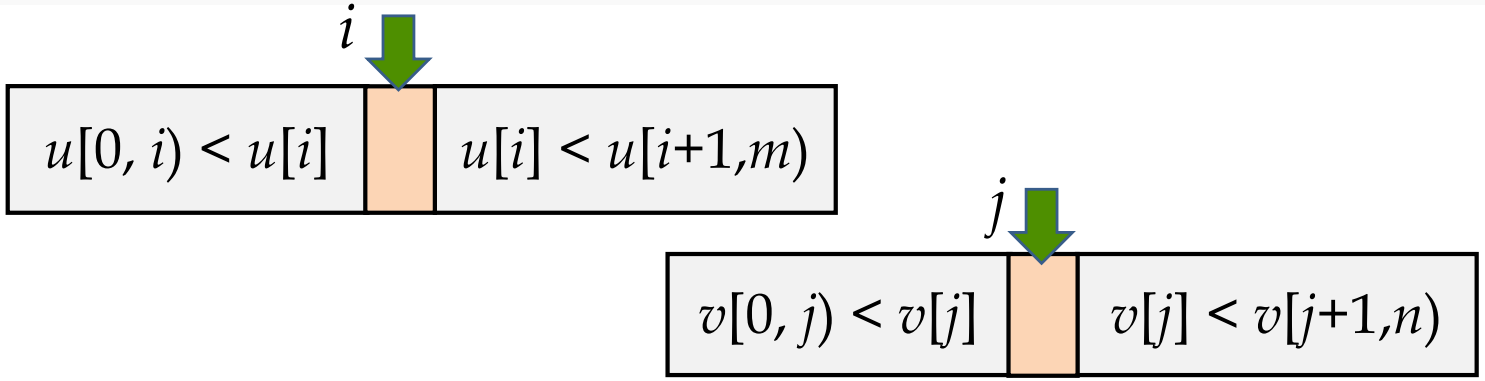
Questa proprietà è necessaria per far vedere che l'invariante si conserva nei programmi precedenti, in quanto:

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

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

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

Cosa possiamo dedurre da **Cr**$(u)$, **Cr**$(v)$ confrontando $u[i]$ e $v[j]$?



# ...qualche buona notizia...

Se $u[i] = v[j]$, da **Cr**($u$) & **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 **PC**($u[0, i)$, $v[j, n)$) = 0 e **PC**($u[i, m)$, $v[0, j)$) = 0 che unito alla formula generale, implica in particolare:

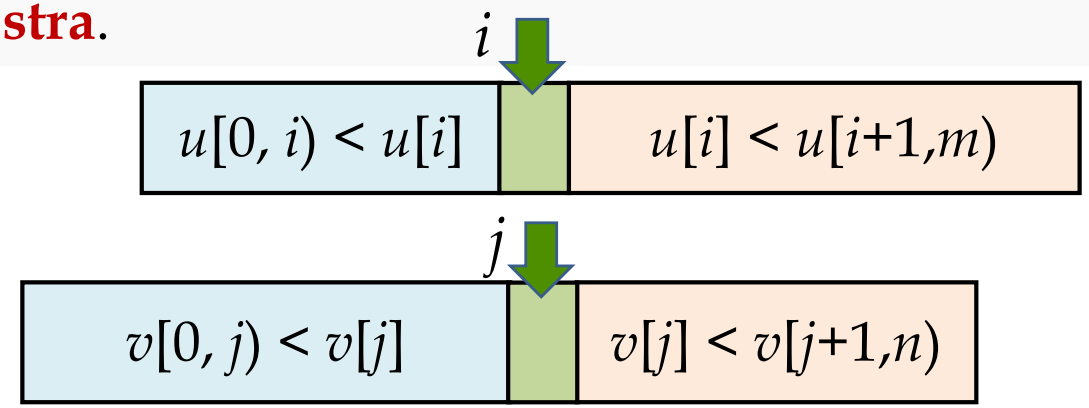
$$\textbf{PC}(u[0, m), v[0, n)) = 1 + \textbf{PC}(u[0, i), v[0, j)) + \textbf{PC}(u[i+1, m), v[j+1, n))$$

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** $c = \textbf{PC}(u[0, i), v[0, j))$. Da cui:

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

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



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

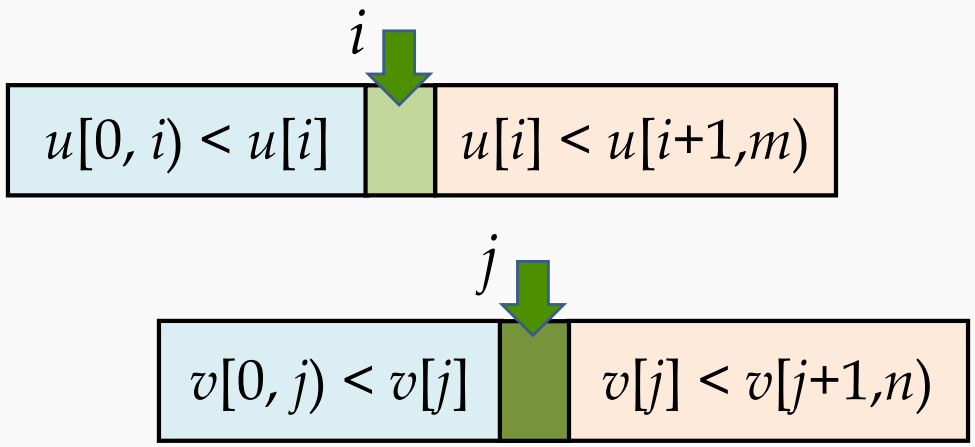
Se $u[i] < v[j]$, posso analogamente dedurre

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

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...

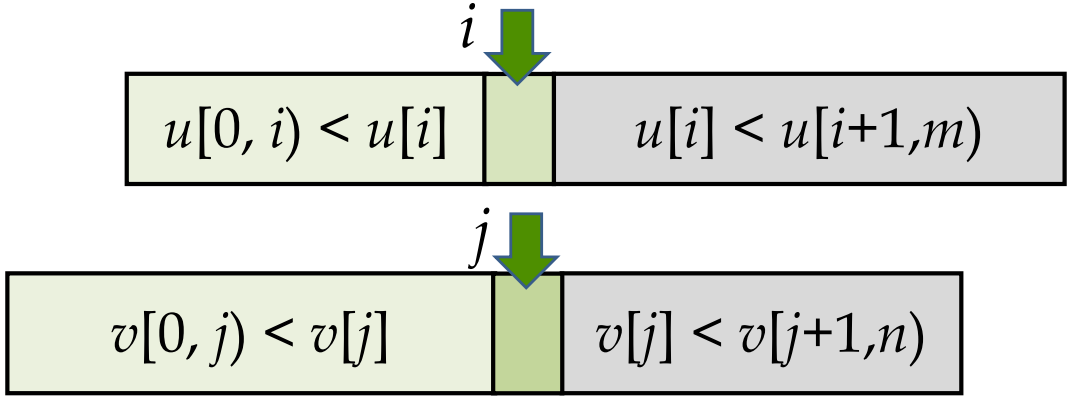
Tuttavia, se sapessimo per qualche motivo che $u[i] > v[0, j)$ avremo ovviamente **PC**$(u[i, m), v[0, j)) = 0$ e quindi il **via libera verso destra** e **simmetricamente** $v[j] > u[0, i)$ ci permetterebbe di gestire il caso simmetrico $u[i] > v[j]$.

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

Ricordiamo che $u[\mathbf{i}] > v[0, j)$ implica $u[\mathbf{i+1}] > v[0, j)$, 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**:

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



# ...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 precendente, 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.

L'esempio più notevole ovviamente è la **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
```