In questa lezione

- **La funzione FONDI che fonde due arrays ordinati in un unico array ordinato**

- **La funzione PARTIZIONE che prende in input un array non ordinato di interi e individua la posizione che dovrebbe assumere un elemento scelto di un array di interi, se fosse ordinato. Per semplicità assumiamo che questo elemento intorno al quale dividere l'array sia l'ultimo o il primo nell'array e lo chiamiamo pivot. Gli elementi minori o uguali del pivot precederanno allora alla fine tutti quelli maggiori. La funzione restituisce la posizione del pivot dopo avervelo spostato.**

N.B. Queste operazioni devono svolgersi localmente, cioè utilizzando una memoria ausiliaria di dimensione costante.

Fondi

Fondi(L,R,A)

Input: L,R ed A sono array di m, n e n+m elementi

prec: L e R sono ordinati, cioè

L[1] ≤ ...≤ L[m-1] e R[1]≤…≤R[n-1]

postc: A contiene gli elementi di L e di R e

A[0] ≤ ...≤ A[m+n-1]

Fondi

Fondi(L,R,A)

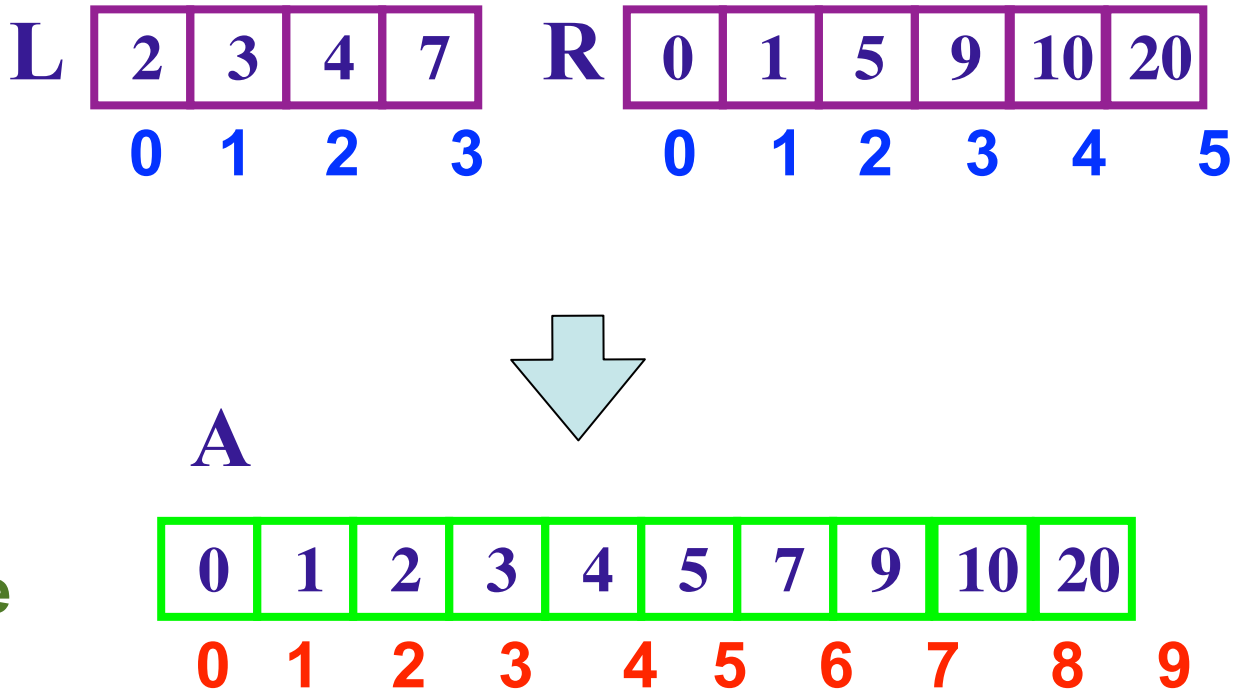
Input: L,R ed A sono array di m, n e n+m elementi

prec: L e R sono ordinati, cioè

L[1] ≤ ...≤ L[m-1] e R[1]≤…≤R[n-1]

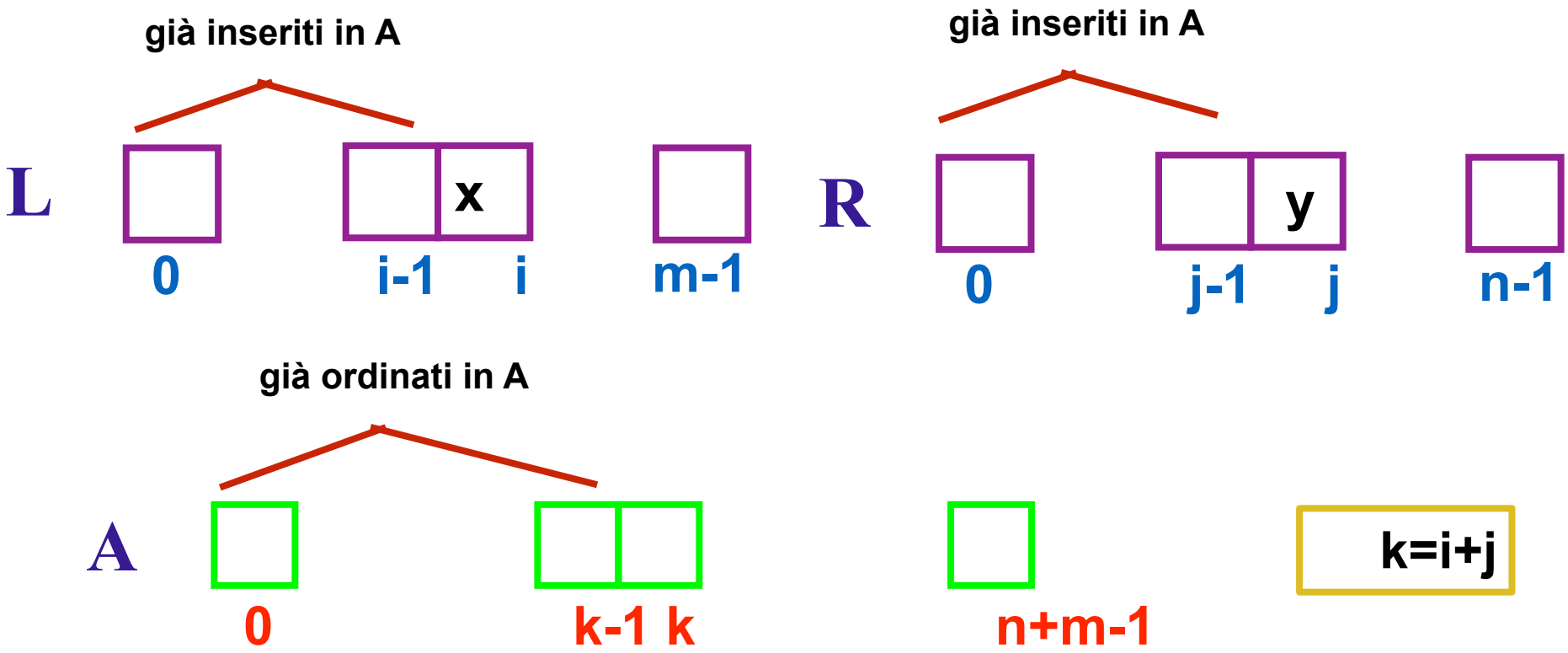
output: A contiene gli elementi di L e di R ordinati,

A[0] ≤ ...≤ A[m+n-1]

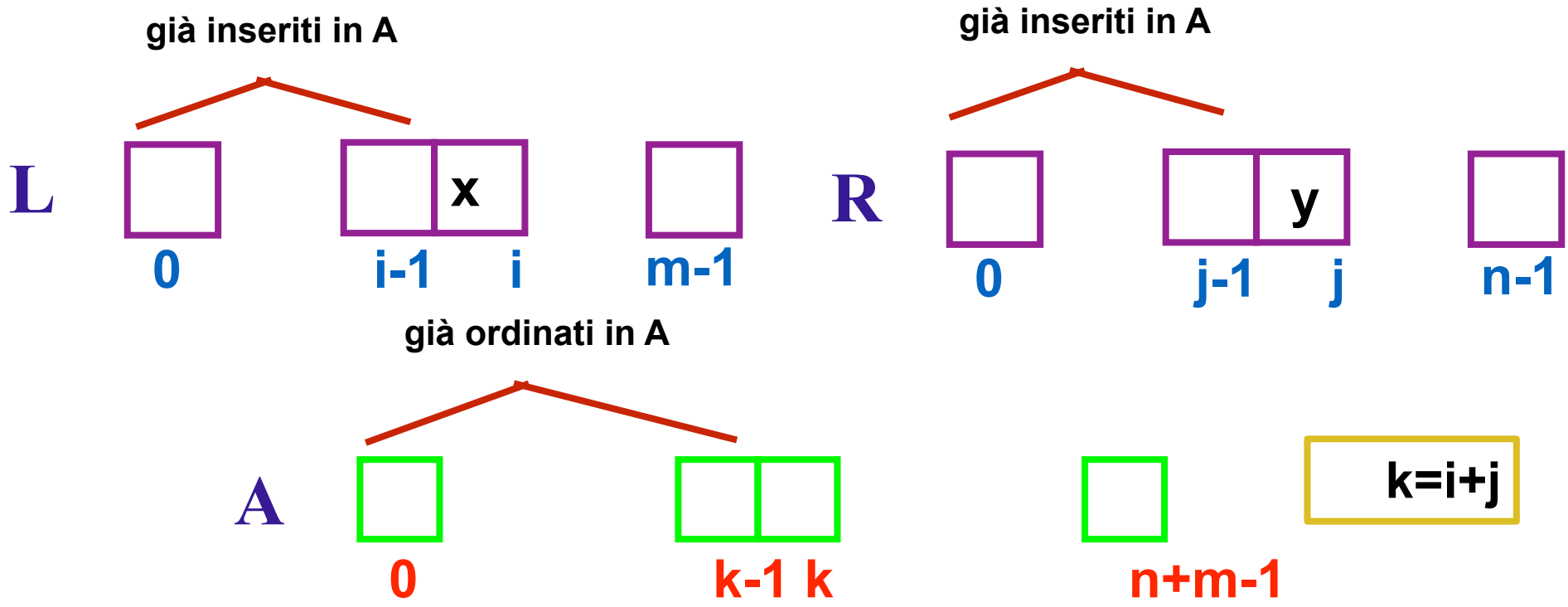


L'idea algoritmica

Per descrivere l'idea ci mettiamo in un passo intermedio in cui immaginiamo di aver già inserito correttamente in A i elementi da L e j elementi da R e ci chiediamo qual'è il prossimo passo.



L'idea algoritmica



Se L[i] ≤ R[j], allora si inserisce L[i] in A[k], altrimenti R[j].
Gli elementi già inseriti sono più piccoli di L[i] e di R[j] e sono ordinati in A. Se si inserisce L[i] si mantiene l'ordinamento:
se in A[k-1] c'è L[i-1], perché L[i-1] ≤ L[i]
se in A[k-1] c'è R[j-1], sappiamo che R[j-1]≤L[i] (perchè è stato scelto al passo precedente) e quindi in entrami i casi A[0]≤ …≤A[k].
Se si inserisce R[j] il ragionamento è analogo.

Fondi


L
2
3
4
7
0 1 2 3
i-1 i
R
0
1
5
9
10
20
0 1 2 3 4 5
j-1 j
A
0
1
2
3
4
5
7
9
10
20
0 1 2 3 4 5 6 7 8 9
k-1 k

Lo pseudocodice

```
Fondi(L,R,A)
Input: L,R ed A  sono array di m, n e n+m elementi
prec: L e R sono ordinati, cioè
L[1] ≤ ...≤ L[m-1] e R[1]≤…≤R[n-1]
postc: A contiene gli elementi di L e di R e A[0] ≤ ...≤ A[m+n-1]
m =  L.size
n = R.size
i=j=k=0
 while  i < m and  j < n   do
l'invariante:  A contiene gli elementi L[0],...,L[i-1] e gli elementi R[0],...,R[j-1],
 in modo che A[0] ≤ A[1] ≤...≤ A[k-1],con k=i+j
          if (L[i] ≤ R[j]) then A[k] = L[i]; i=i+1
              else A[k] = R[j]; j=j+1
          k=k+1
  while  i<m  do % uscita per j=n, copia eventuali elementi rimanenti di L,
       A[k] = L[i]; k=k+1; i=i+1
  while  j<n  do  % uscita per i=m, copia eventuali elementi rimanenti di R,
     A[k] = R[j]; k=k+1; j=j+1
```

Tempo di esecuzione

Fondi(L,R,A)

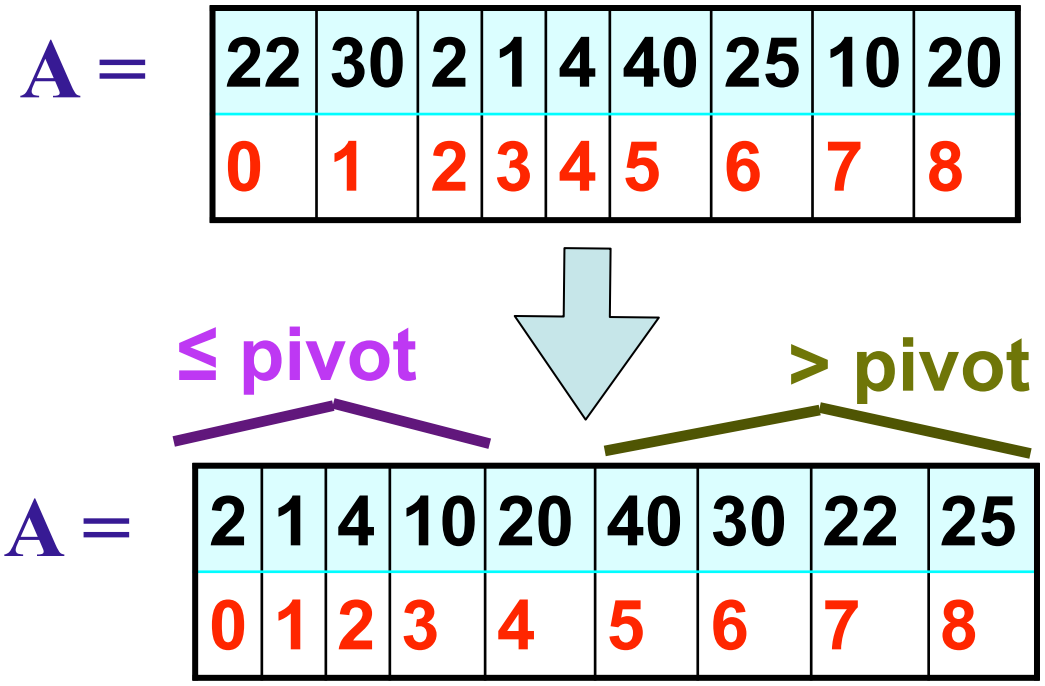
Input: L,R ed A sono array di m, n rispettivamente n+m elementi
prec: L e R sono ordinati, cioè
L[1] ≤ ...≤ L[m-1] e R[1]≤…≤R[n-1]
postc: A contiene gli elementi di L e di R e A[0] ≤ ...≤ A[m+n-1]

```
m =  L.size
n = R.size
i=j=k=0
while  i<m and  j<n   do
       if L[i] ≤ R[j]
              then A[k] = L[i]
                   i=i+1
              else A[k] = R[j]
                   j=j+1
       k=k+1
while  i<m  do
      A[k] = L[i]
       k=k+1
       i=i+1
while  j<n  do
      A[k] = R[j]
       k=k+1
       j=j+1
```

Le istruzioni all'interno dei cicli sono eseguite in tempo costante. Il primo ciclo viene eseguito al più n + m volte, se per esempio gli elementi vengono scelti alternativamente da uno e dall'altro array e si svuotano contemporaneamente, e come minimo min{n,m} volte, quando l'array con meno elementi contiene elementi tutti più piccoli del primo dell'altro array. Il secondo o il terzo ciclo fanno sì che vengano copiati tutti gli elementi rimanenti. Quindi concludiamo che la funzione ha un tempo di esecuzione asintotico in Θ(n+m).

Partizione

Bisogna trovare l'indice che individua la posizione "giusta" per il pivot, l'ultimo elemento, 20 nell'esempio, e disporre gli elementi minori o uguali del pivot prima e quelli maggiori dopo il pivot.

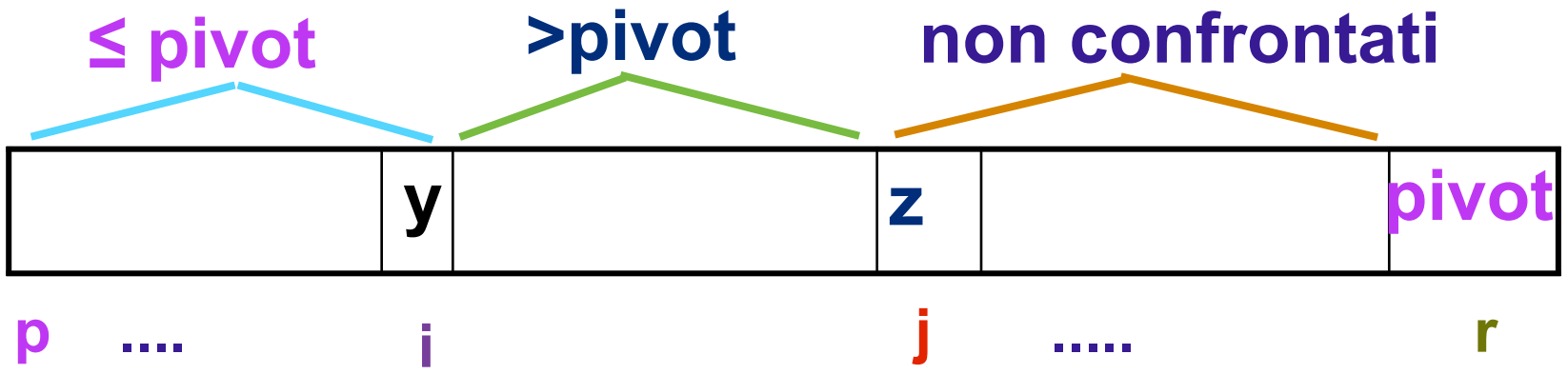


4 è l'indice giusto per 20: gli elementi di indice 0 fino a 3 sono più piccoli e quelli da 4 a 8 sono più grandi di 20

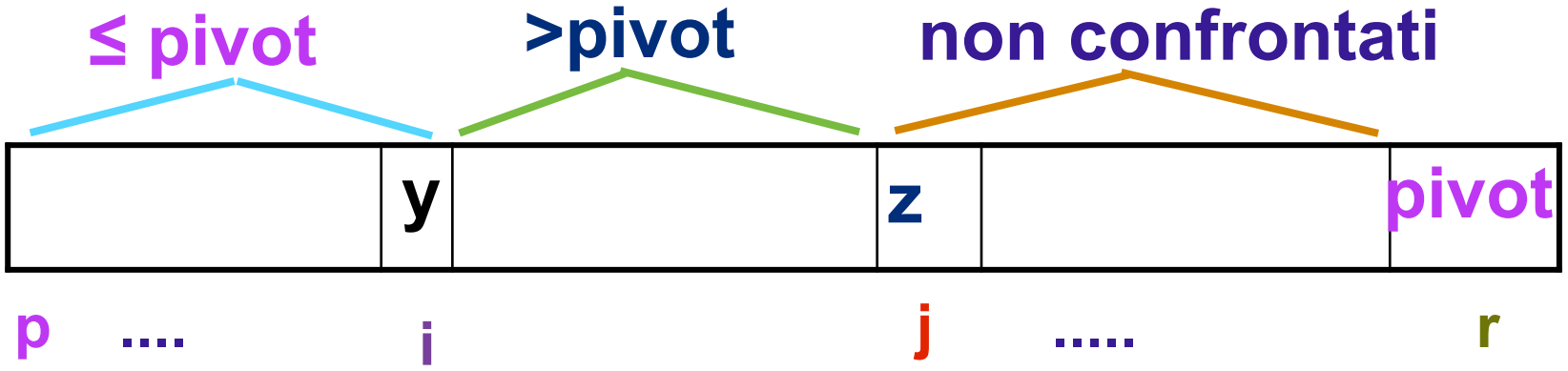
Partizione: approccio semplice

Passo intermedio.

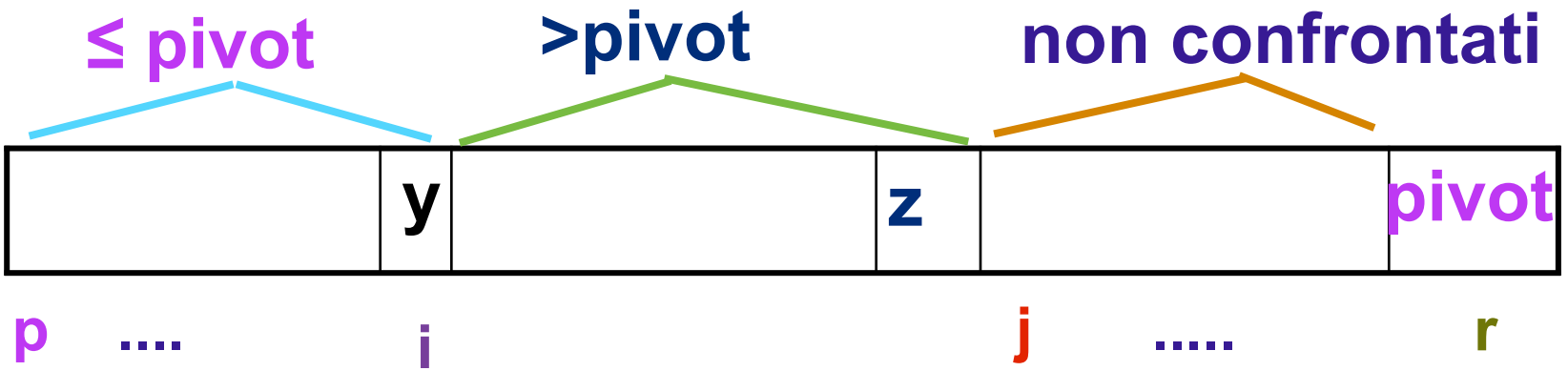
Se si scorre l'array da sinistra a destra con un indice j, ci si chiede cosa fare quando si considera l'elemento A[j], nell'ipotesi che le operazioni eseguite fino a questo momento abbiano correttamente separato gli elementi tra minori o uguali del pivot e maggiori:



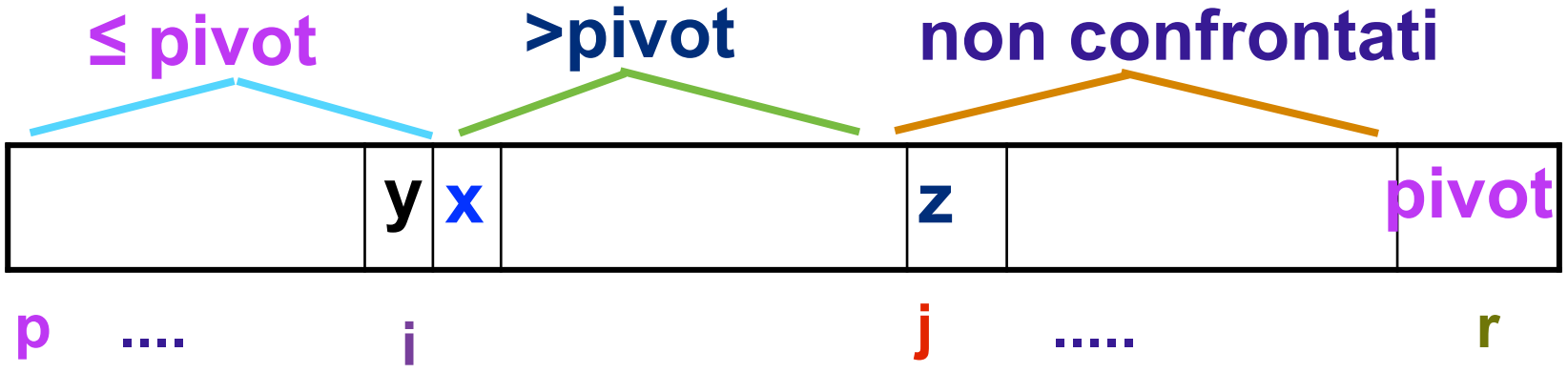
Partizione: caso 1



Se A[j] > pivot basta incrementare j per controllare l'elemento successivo.

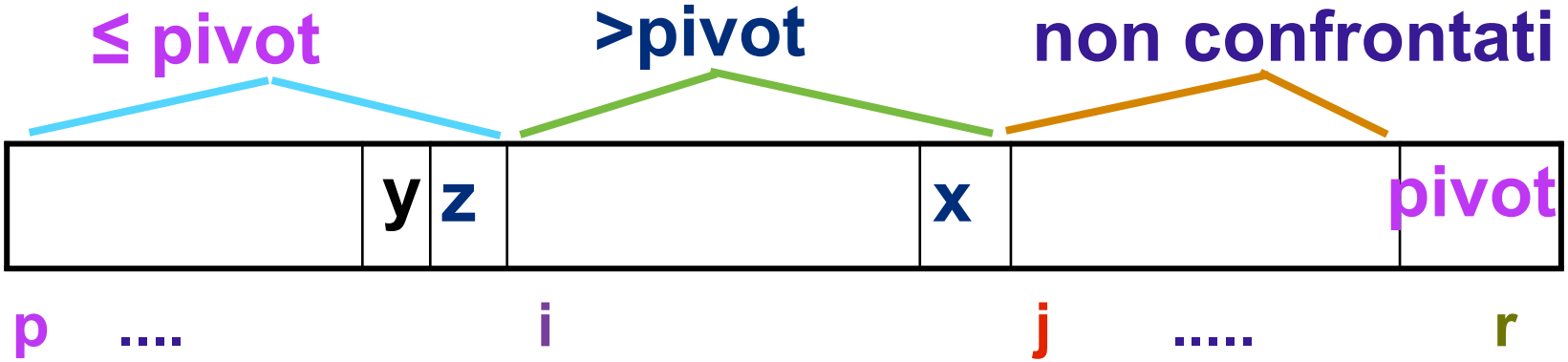


Partizione: caso 2



Se A[j] ≤ pivot , allora deve A[j] deve ampliare la "zona" dei più piccoli. Per ottenere questo basta scambiarlo con A[i+1]

e incrementare i e j.



Inizializzazione indici

Vogliamo che in ogni esecuzione del ciclo sia vero che

se p ≤ k ≤ i allora A[k] ≤ pivot

se i +1≤ k ≤ j-1 allora A[k] > pivot

se k = r allora A[k] = pivot

i= p-1 e j = p **....** **r**

j è il nostro indice di scorrimento dell'array dunque inizialmente j = p visto che dobbiamo confrontare tutti gli elementi escluso il pivot. Allora i = p - 1, visto che non ci sono elementi minori del pivot e così l'intervallo di indici tra p e i è vuoto.

Pseudocodice di Partizione

```
Partizione(A,p,r)
 pivot = A[r]
 i = p-1     All'inizio non ci sono elementi ≤ pivot
 for j = p  to r-1 do  e nemmeno > pivot!
          if  A[j] ≤ pivot then
                  i =  i+1
                  scambia A[i] con A[j]
 scambia A[i+1] con A[r]
i+1 è la posizione del pivot
 return i+1
```

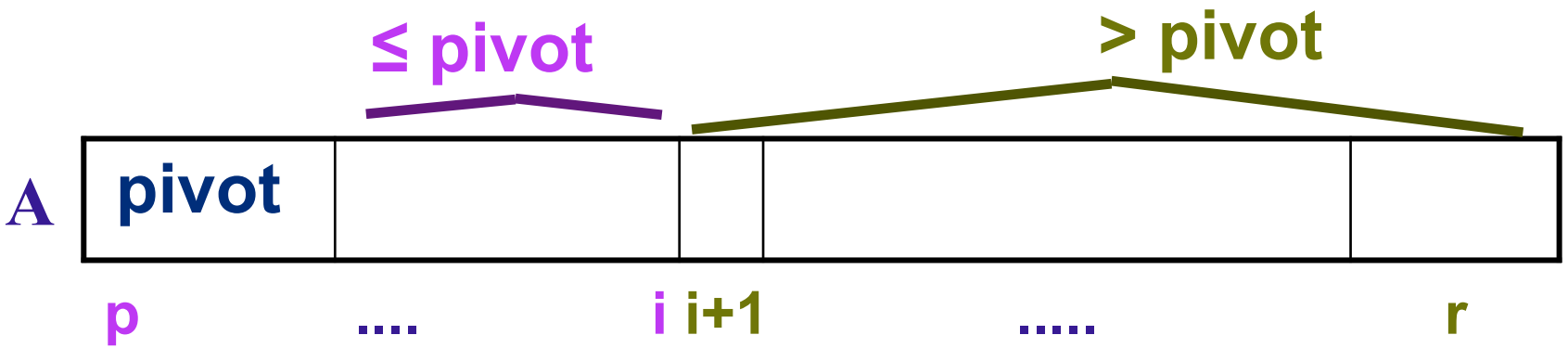
Tempo di esecuzione

```
Partizione(A,p,r)
 pivot = A[r]
 i = p-1     All'inizio non ci sono elementi ≤ pivot
 for j = p  to r-1 do  e nemmeno > pivot!
          if  A[j] ≤ pivot then
                  i =  i+1
                  scambia A[i] con A[j]
 scambia A[i+1] con A[r]
i+1 è la posizione del pivot
  return i+1
```

Le istruzioni all'interno del ciclo sono eseguite in tempo costante, il ciclo è eseguito r- p volte quindi il tempo di esecuzione in tutti i casi è in Θ(n), dove n = r - p +1 è il numero degli elementi in A.

La partizione di un array: l'algoritmo di Hoare

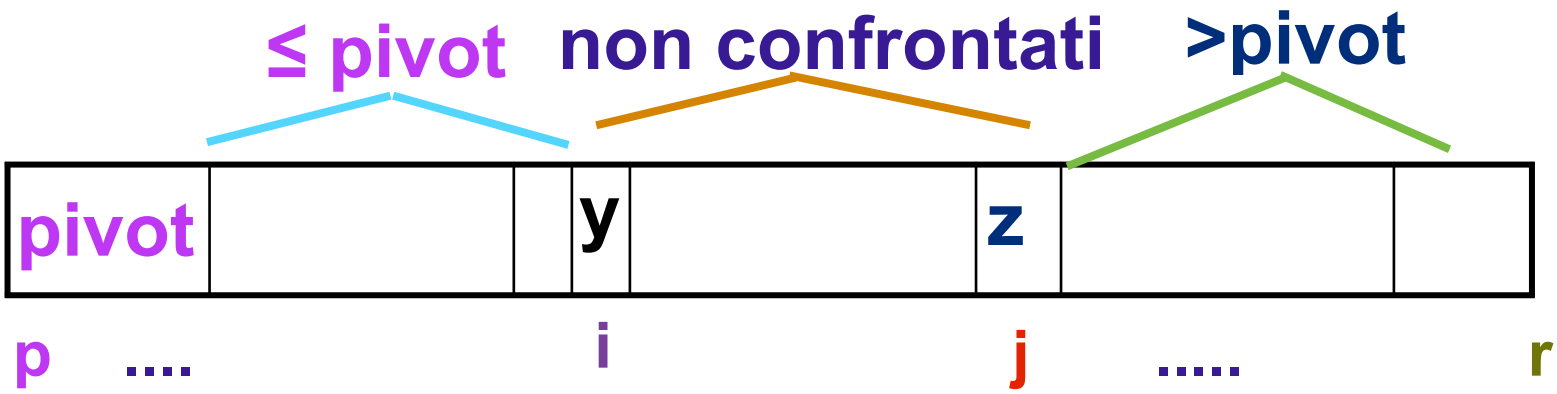
Ora il pivot è il primo elemento. Bisogna trovare l'indice che individua la posizione "giusta" per il pivot, e disporre gli elementi minori o uguali del pivot prima e quelli maggiori dopo il pivot.



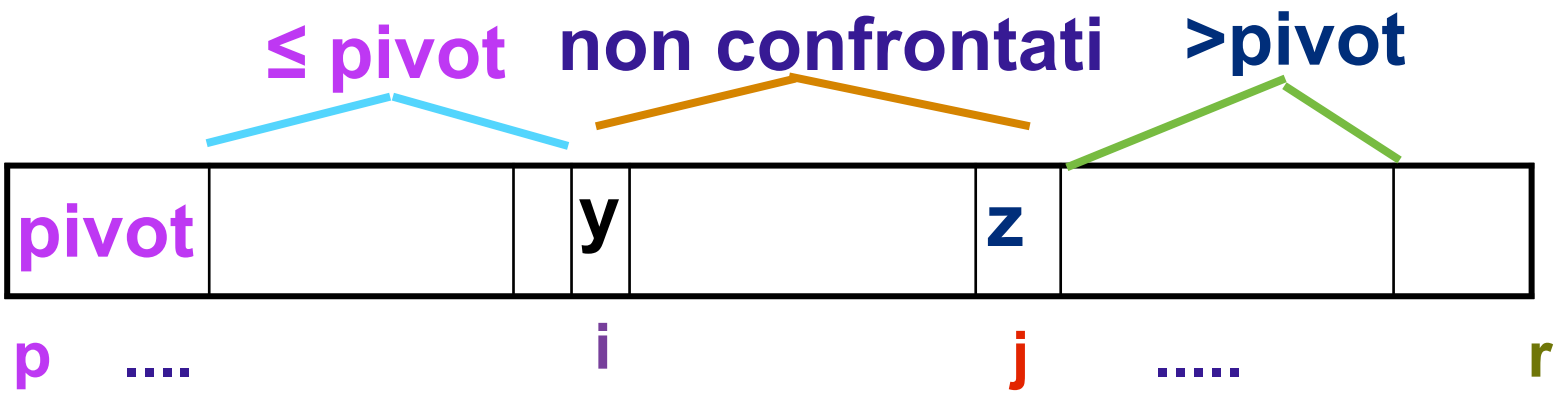
L'idea di Hoare è di scorrere l'array sia da sinistra verso destra che da destra verso sinistra. Sia i l'indice di scorrimento da sinistra e j quello da destra. Si incrementa i fintanto che gli elementi sono minori del pivot, e analogamente si decremento j fintanto che gli elementi sono maggiori. Ora l'elemento A[i] ≤pivot e A[j]> pivot, lo scambio dei due elementi e l'incremento degli indici amplia entrambe le zone dei minori o uguali e dei maggiori e si può ricominciare con i confronti fino ad esaurimento degli elementi.

La partizione di Hoare, passo intermedio

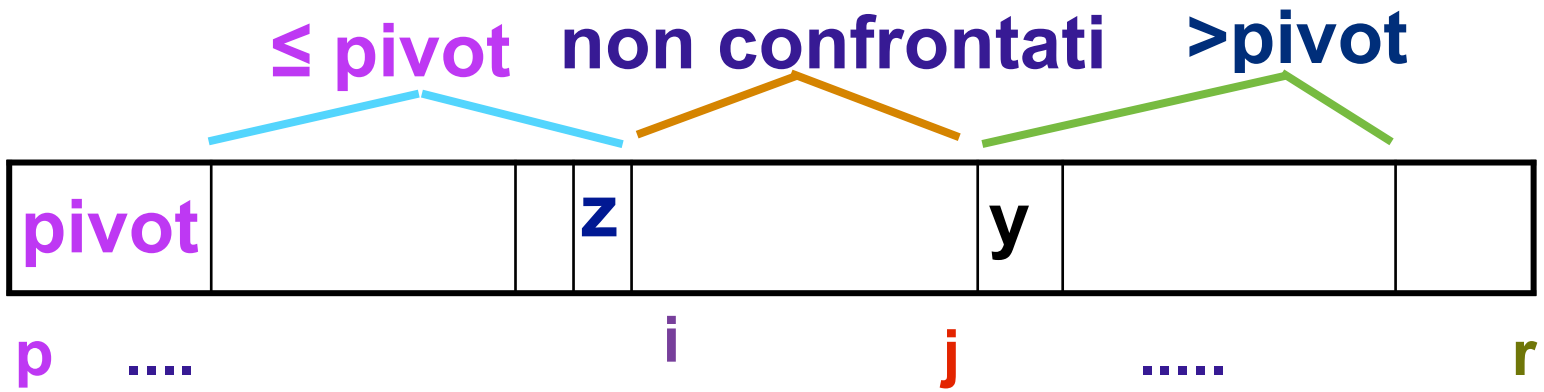
Immaginiamo di trovarci in un passo intermedio e che le istruzioni eseguite fino a ora correttamente abbiamo individuato i maggiori del pivot a destra e i minori o uguali a sinistra nell'array in due zone delimitate da i a sinistra e da j a destra. Poiché i incrementa fintanto che gli elementi sono minori o uguali del pivot, e j si decrementa fintanto che gli elementi sono maggiori, vuol dire che A[i] > pivot e A[j] ≤ pivot:



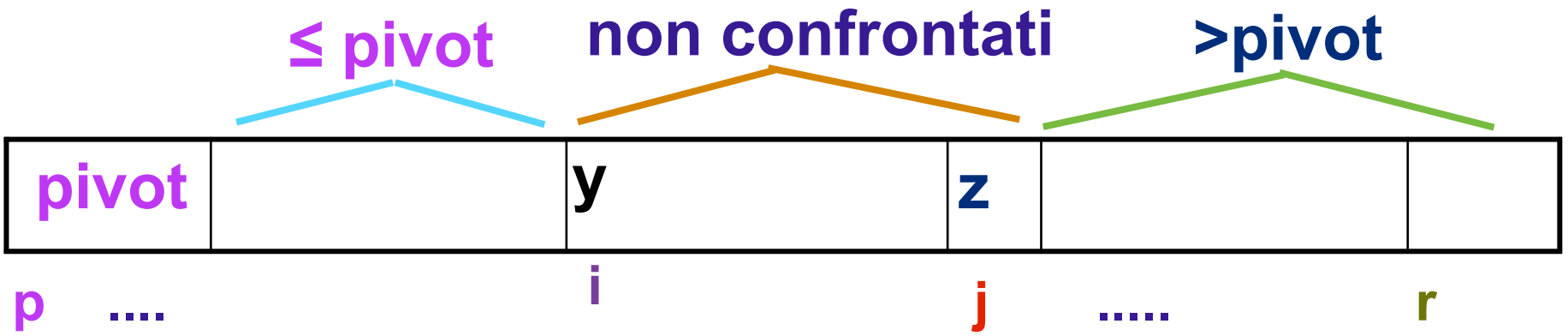
La partizione di un array: l'idea di Hoare, passo 2



Si scambia A[i] e A[j], si incrementa i e si decrementa j e si ha:



Pseudocodice di Partizione2



Le istruzioni da inserire nel ciclo devono realizzare:

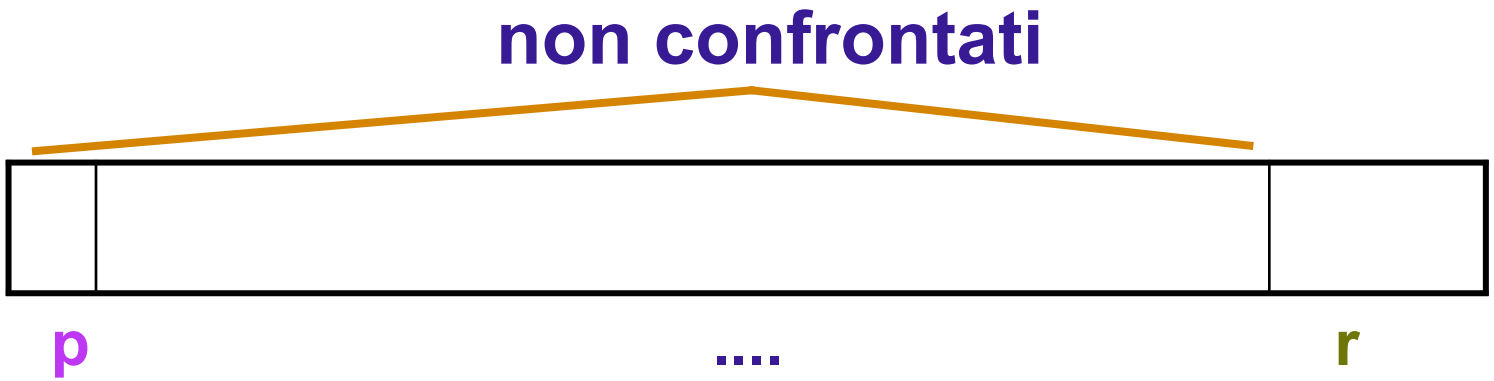
finchè A[i] ≤ pivot do i++ Da questo ciclo si esce con A[i]>pivot

finchè A[j] > pivot do j- - Da questo ciclo si esce con A[j]≤pivot

scambia A[i] con A[j]

Il ciclo viene eseguito finché ci sono elementi da confrontare, quindi finché i≤j

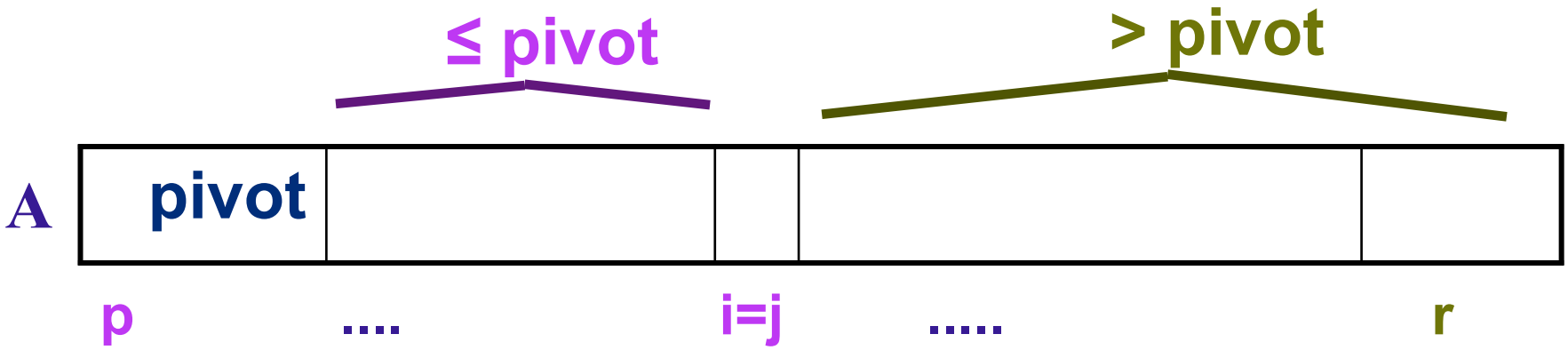
Inizializzazione degli indici i e j



Quelli da confrontare sono tutti gli elementi tra p+1 e r, visto che nella prima posizione p c'è il pivot:

i = p+1 **j = r**

L'indice per il pivot



Abbiamo elementi da confrontare fino a che i≤j.

L'ultimo elemento da confrontare sarà per i = j

e se A[i] ≤ pivot si incrementa i (per cui i>j) e quindi A[j] è l'ultimo dei minori o uguali al pivot

altrimenti A[j] > pivot e allora si decrementa j (per cui i>j) e quindi j è la posizione dell'ultimo minore o uguale al pivot.

Dunque j è la posizione finale per il pivot.

Psudocodice

```
PartizioneHoare(A,p,r)
pivot = A[p]
i = p+1  All'inizio non ci sono elementi ≤ pivot
j  = r e nemmeno > pivot!
while i≤j   do
        while  A[j] > pivot do j = j-1
                    %qui si esce se A[j]≤pivot
        while  A[i] ≤ pivot and i≤j do i = i+1
        if (i<j) scambia A[i] con A[j]
                 j = j-1
                 i = i+1
scambia A[j] con A[p]
return j
```

La versione originale di Hoare non è esattamente questa.

Analisi tempo di esecuzione

```
PartizioneHoare(A,p,r)
pivot = A[p]
i = p+1  All'inizio non ci sono elementi ≤ pivot
j  = r e nemmeno > pivot!
while i≤j   do
        while  A[j] > pivot do j = j-1
                    %qui si esce se A[j]≤pivot
        while  A[i] ≤ pivot and i≤j do i = i+1
        if (i<j) scambia A[i] con A[j]
                 j = j-1
                 i = i+1
scambia A[j] con A[p]
return j
```

Ogni elemento è confrontato una sola volta con il pivot, quindi il tempo è in Θ(n), se n è il numero degli elementi in A.