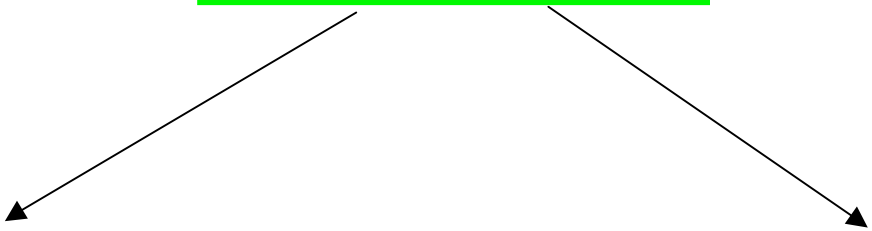
Divide et Impera



Quicksort

Charles Antony Richard Hoare
(senior researcher with Microsoft Research in Cambridge)
Computer Jurnal 5,1,1962

Mergesort

John von Neumann(1903-1957)
Nel 1944, il suo rapporto interno "First Draft of a Report on the EDVAC" contiene tra l'altro, il mergesort come programma di ordinamento

Quicksort

- Esempio di Divide et Impera

  – la fase di partizione

    Dividi la lista degli elementi in due et Impera sistemando al posto giusto nell'ordine un elemento per chiamata

Quicksort

- **La fase di partizionamento**
  - **Scegli un pivot**
  - **Trovane la posizione giusta nella lista**
    - **tutti gli elementi a sinistra sono minori**
    - **tutti gli elementi a destra sono maggiori o uguali**

Quicksort

– Applica lo stesso algoritmo alle due parti

Quicksort: *Pseudocodice*

```
void qSort(int a[], int start, int end)
{ int p;
if (start >= end) return;
/*la ricorsione termina  quando c'è al più un elemento*/
```

dividi il vettore *a* in due sottovettori, i modo tale che la prima metà contenga gli elementi più piccoli del pivot, l'altra i più grandi e il pivot si trovi nella giusta posizione p nel vettore ordinato

Divide

```
qSort(a,start, p-1);
qSort(a,p+1,end);
```

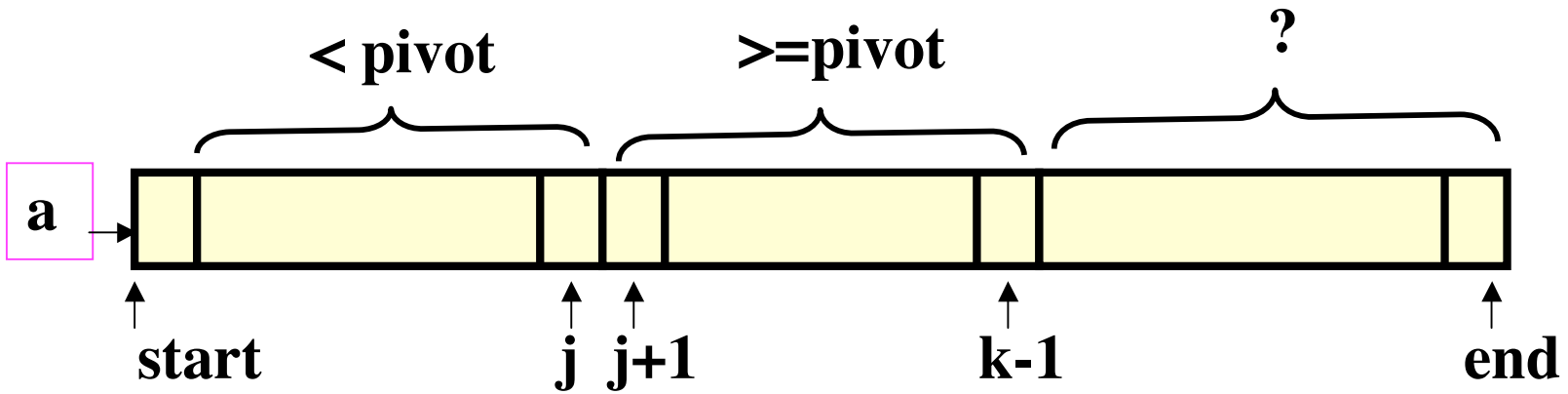
Impera

```
}
```

Quicksort

```
void qSort1(int a[],int start, int end)
 /* ordina in ordine crescente un vettore di interi
* prec: (a !=NULL)
* postc : a[i] <= a[i+1], per start<=i<end.*/
{ int m;
if (start < end)
/* c'è  più  di un elemento*/
   {m = partition(a,start,end); /*m è la posizione del pivot*/
/*Qui (a[start],…,a[m-1] < a[m]<= a[m+1],…,a[end]*/
    qSort1(a,start, m-1);
    qSort1(a,m+1,end);
  }
}
```

Come realizzare il partizionamento? Prima di tutto prendiamo come pivot il primo elemento, per semplicità.
Una prima idea:
scorro l'array confrontando gli elementi con il pivot e sistemando all'inizio tutti i più piccoli e a seguire i più grandi:

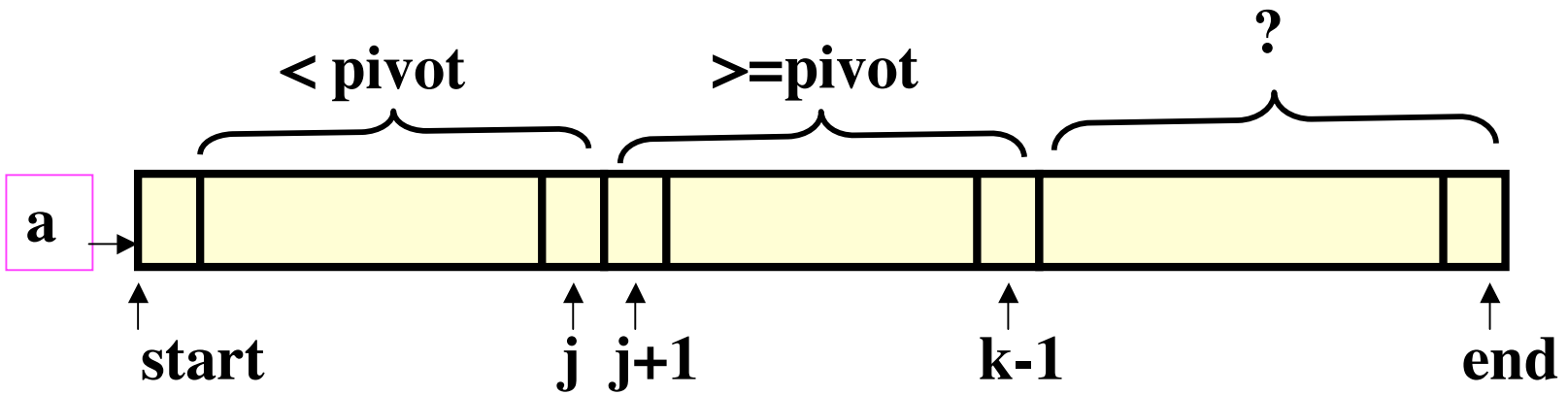


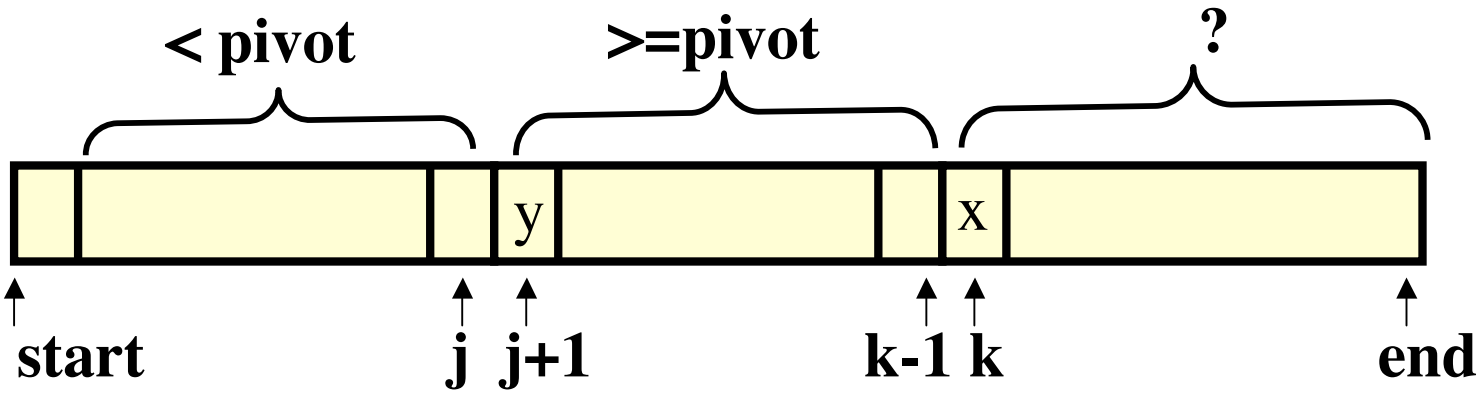
Gli indici j e k servono per individuare le due "zone": quella dei più piccoli e quella dei più grandi,
a[k] è il prossimo elemento da confrontare con il pivot e "sistemare" in conseguenza

pivot = a[start]

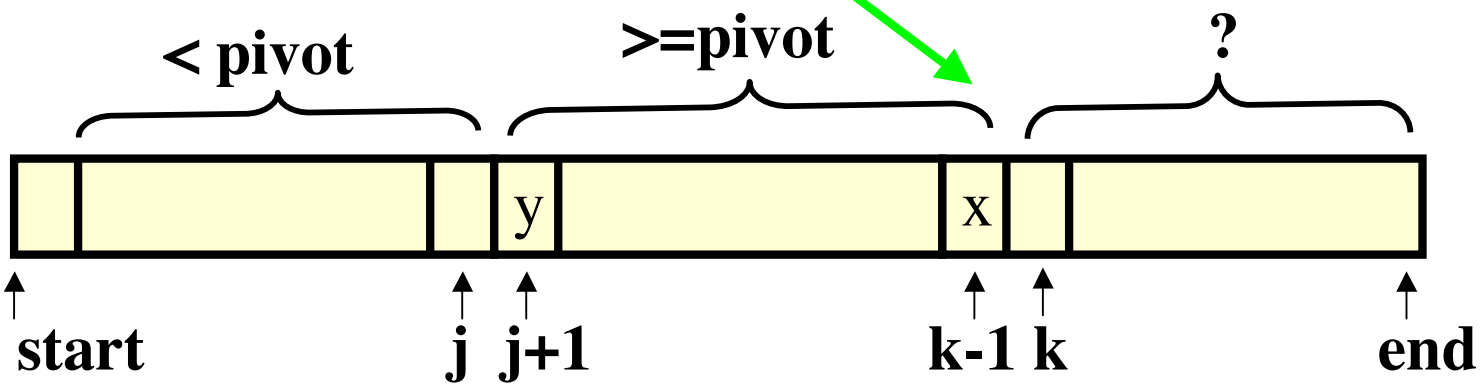
INVARIANTE di ciclo :
un'asserzione che deve essere vera all'entrata per ogni esecuzione del ciclo. Nello specifico, l'invariante del ciclo di partizionamento:

(a[start+1],…,a[j] < pivot) &&
(a[j+1],…,a[k-1] >= pivot)



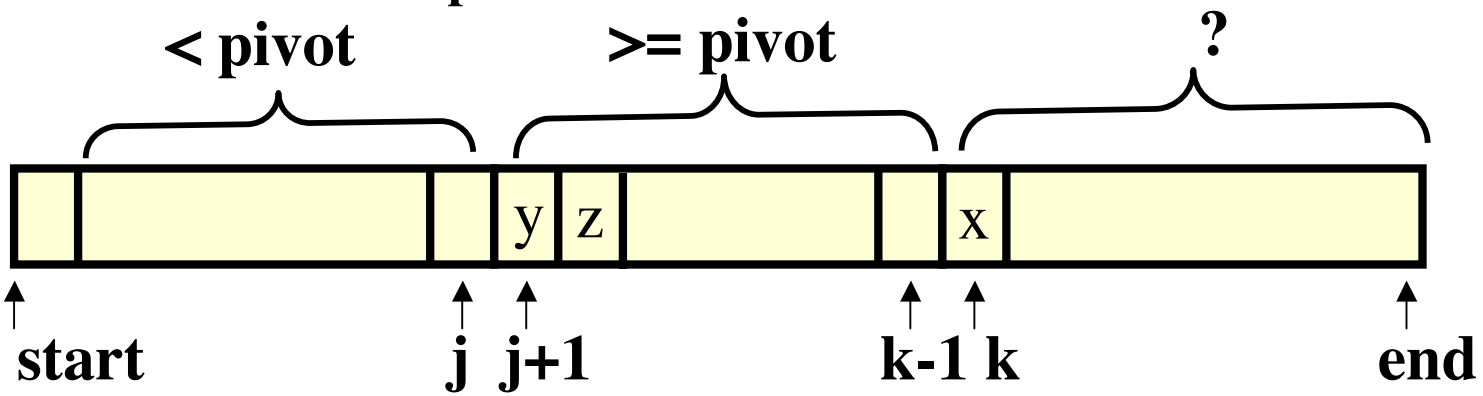


Se a[k] =x >= pivot incrementa k

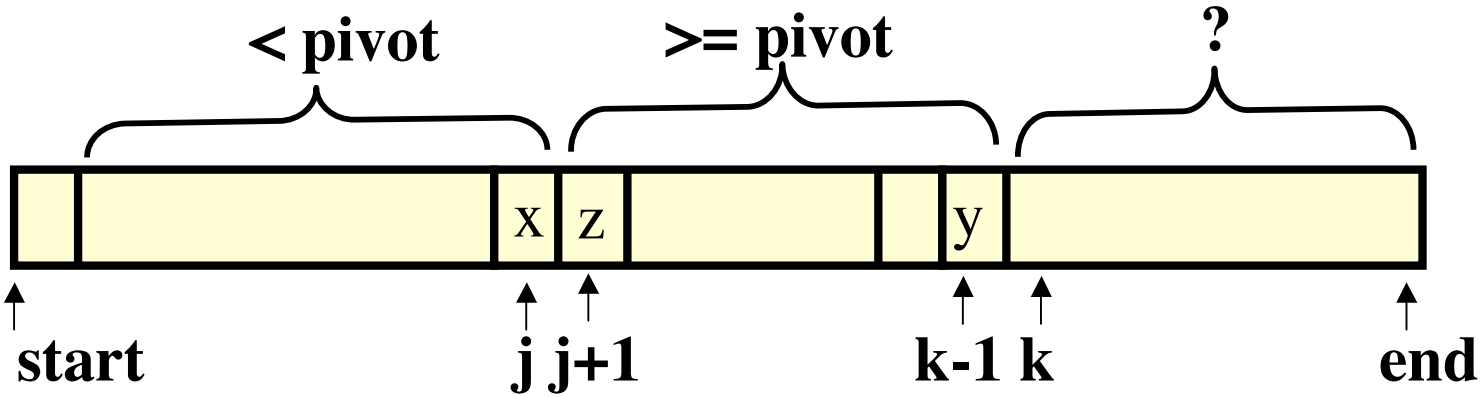


e l'invariante è ancora vero

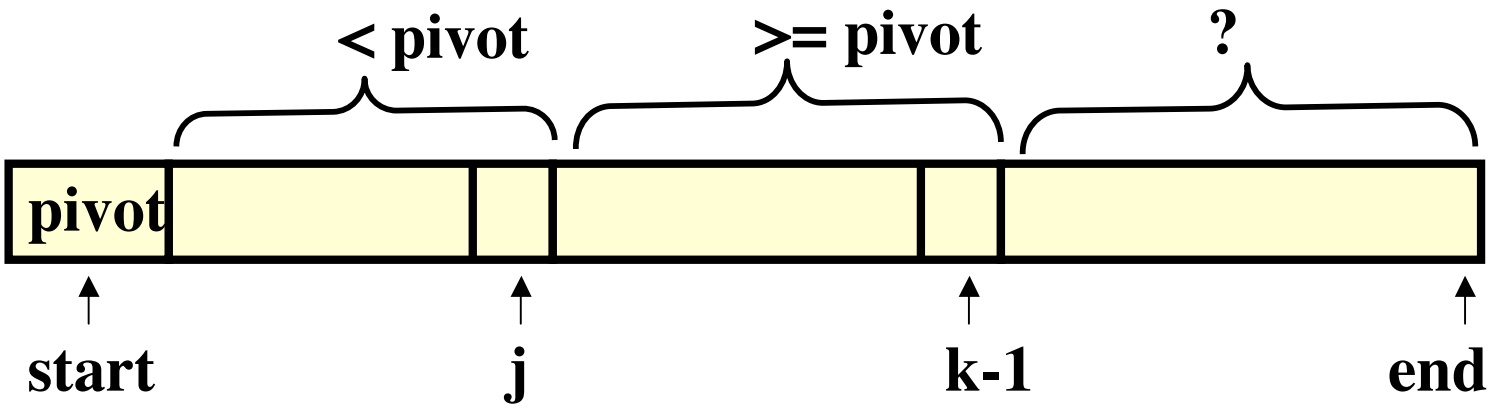
L'invariante del ciclo di partizionamento:



Se a[k] = x < pivot, scambia a[j+1] = y con a[k] = x e incrementa j e k

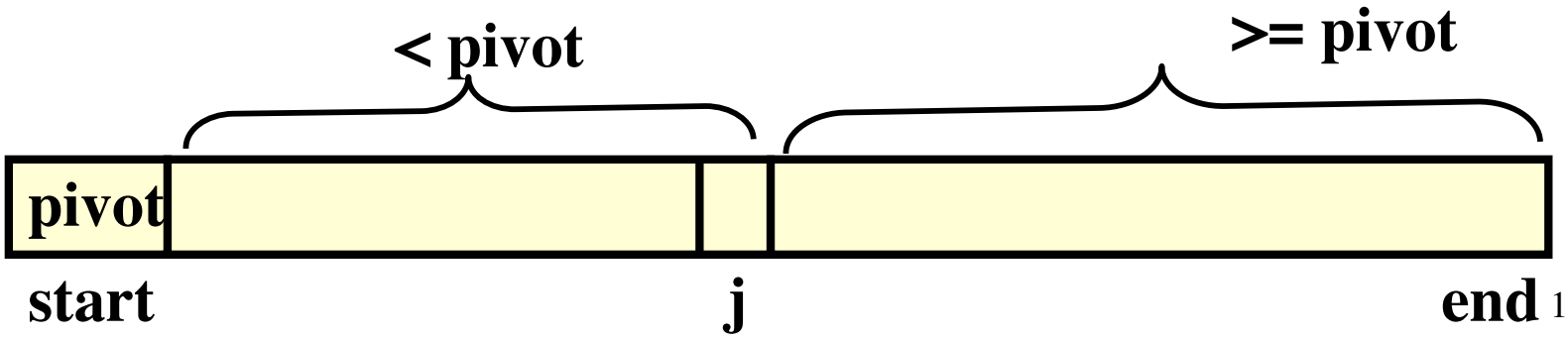


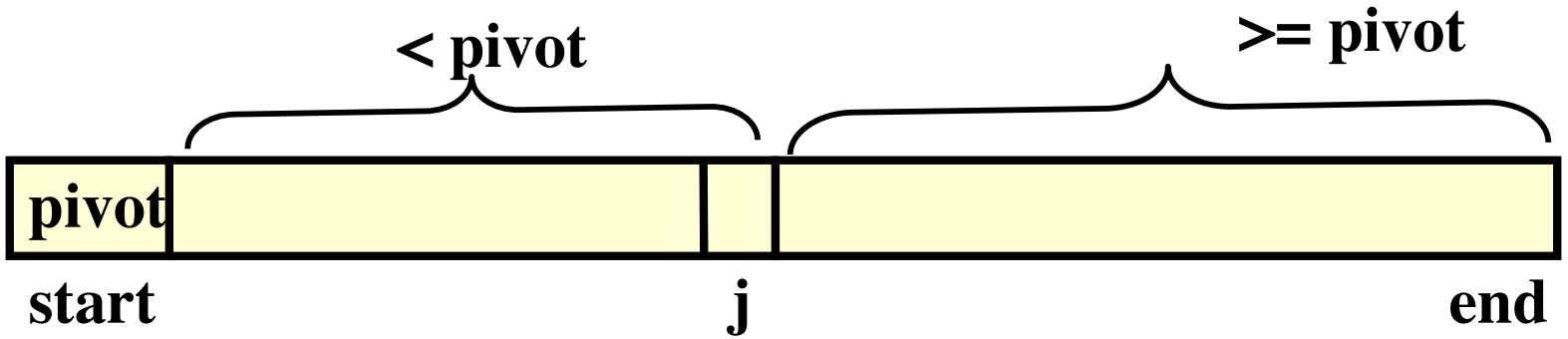
e l'invariante è ancora vero




```
j = start; /* il pivot è a[start] */
for  (k= start+1;k <= end; k++)
/* invariante: (a[start+1],…,a[j] < a[start])&&
(a[j+1],…,a[k-1] >= a[start])*/
          if (a[k] < a[start]) scambio(&a[++j],&a[k]);
```

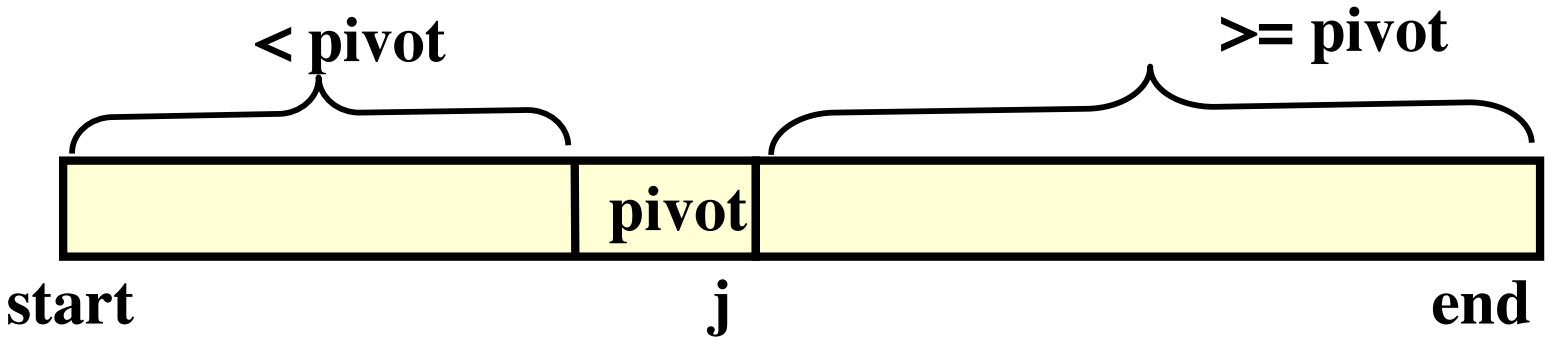
Alla fine del ciclo for di partizionamento:





Con un ulteriore scambio:

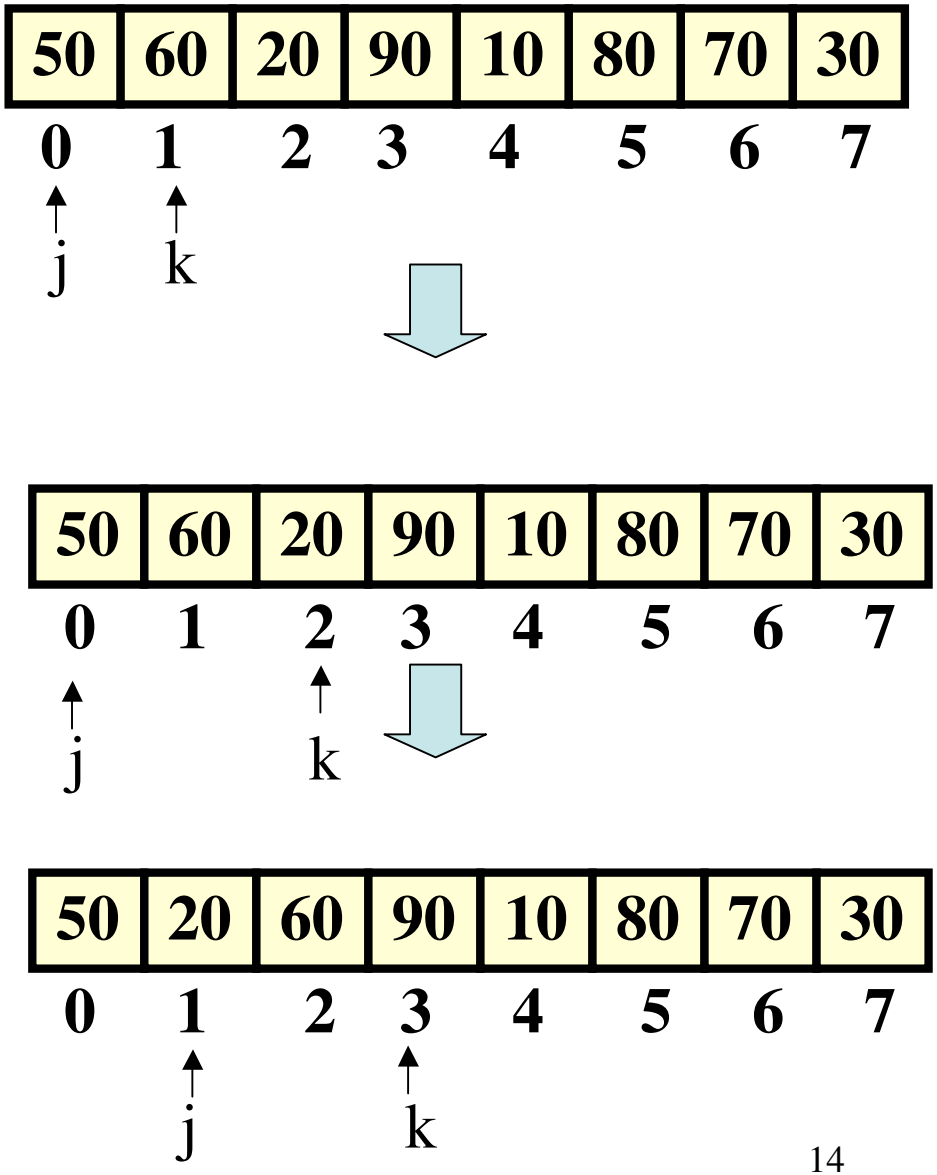
```
scambio(&a[start],&a[j]);
```



```
int partition1(int *a, int in, int fin)
 /* divide il vettore a in due sottovettori
il primo con gli elementi da start a j-1 e il secondo
con quelli da j+1 a end e in modo tale che
(P)(a[in],…,a[j-1] < a[j]<= a[j+1],…,a[fin]*/
postc: restituisce l'indice j tale che valga P*/
{ int j = in, k;
for (k = in+1; k <= fin; k++)
/* invariante: (a[in+1],...,a[j] < a[in]) &&
(a[j+1],...,a[k-1] >= a[in]) */
   if  (a[k] < a[in]) scambia(&a[++j],&a[k]);
scambia(&a[j],&a[in]);
return j;
}
```

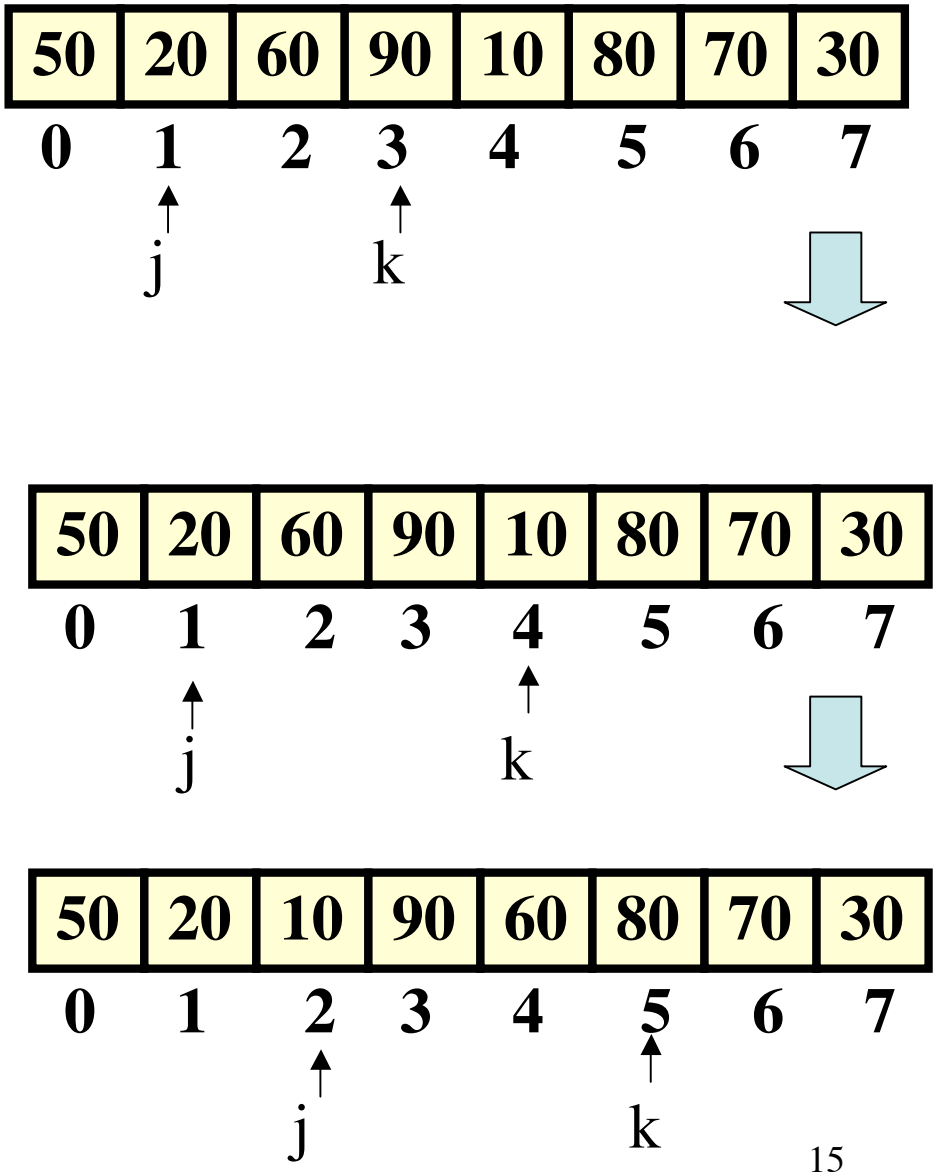
Chiamata partition1(a,0, 7);

```
{ int j = in, k;
for (k = in+1; k <= fin; k++)
/* invariante:
(a[in+1],...,a[j] < a[in]) &&
(a[j+1],...,a[k-1] >= a[in]) */
   if  (a[k] < a[in])
    scambia(&a[++j],&a[k]);
scambia(&a[j],&a[in]);
return j;
}
```



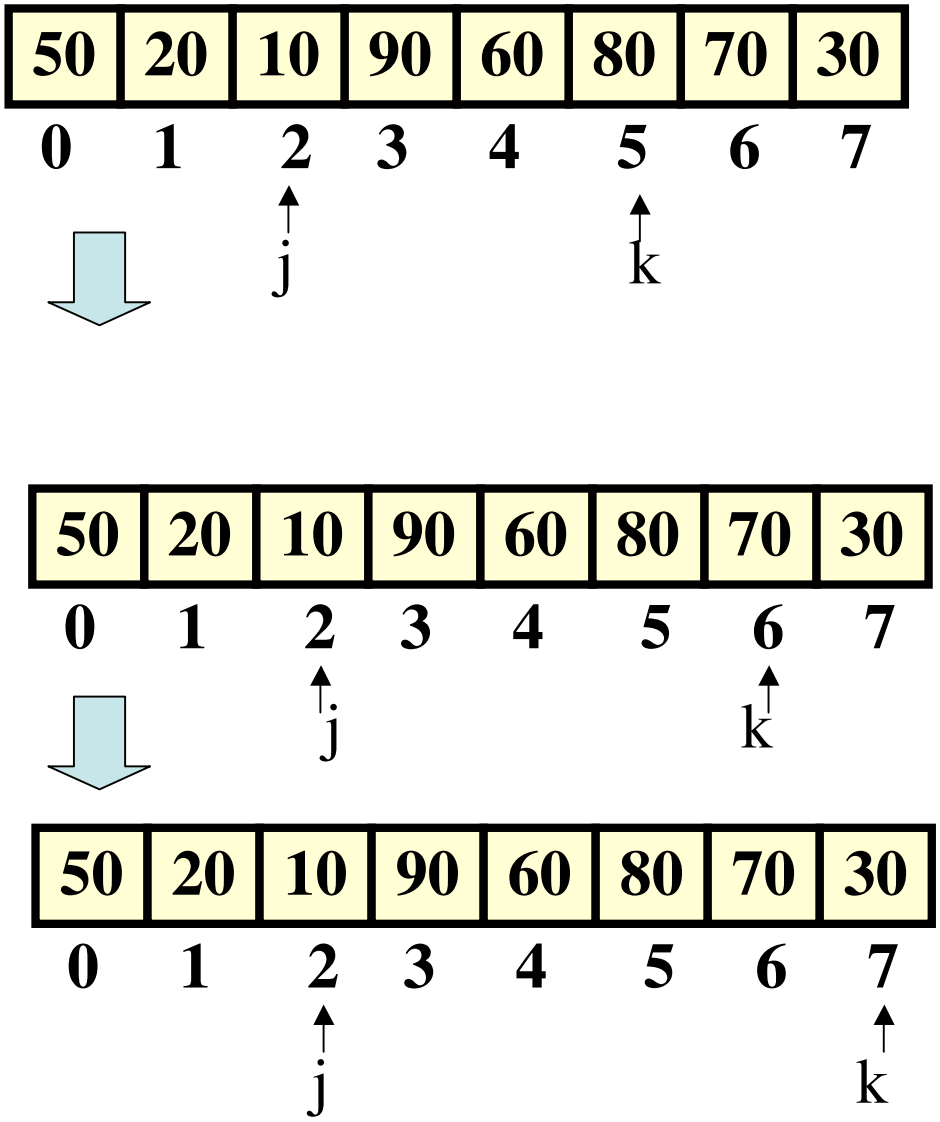
Quicksort - partizione1

```
{ int j = in, k;
for (k = in+1; k <= fin; k++)
/* invariante:
(a[in+1],...,a[j] < a[in]) &&
(a[j+1],...,a[k-1] >= a[in]) */
   if  (a[k] < a[in])
    scambia(&a[++j],&a[k]);
scambia(&a[j],&a[in]);
return j;
}
```

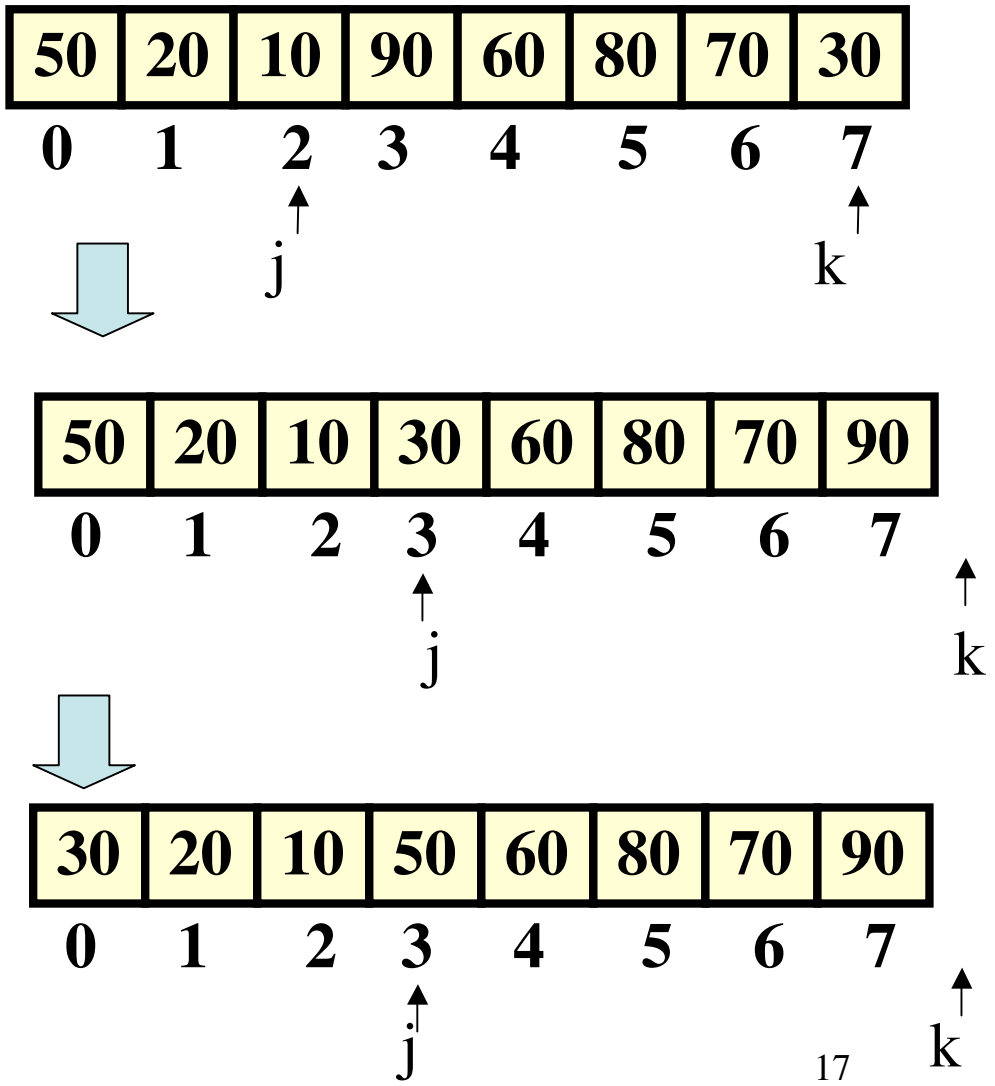


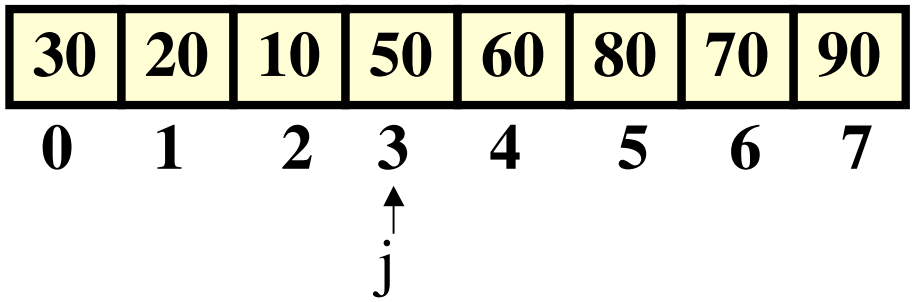
Quicksort - partizione1

```
{ int j = in, k;
for (k = in+1; k <= fin; k++)
/* invariante:
(a[in+1],...,a[j] < a[in]) &&
(a[j+1],...,a[k-1] >= a[in]) */
   if  (a[k] < a[in])
    scambia(&a[++j],&a[k]);
scambia(&a[j],&a[in]);
return j;
}
```



```
{ int j = in, k;
for (k = in+1; k <= fin; k++)
/* invariante:
(a[in+1],...,a[j] < a[in]) &&
(a[j+1],...,a[k-1] >= a[in]) */
   if  (a[k] < a[in])
    scambia(&a[++j],&a[k]);
scambia(&a[j],&a[in]);
return j;
}
```

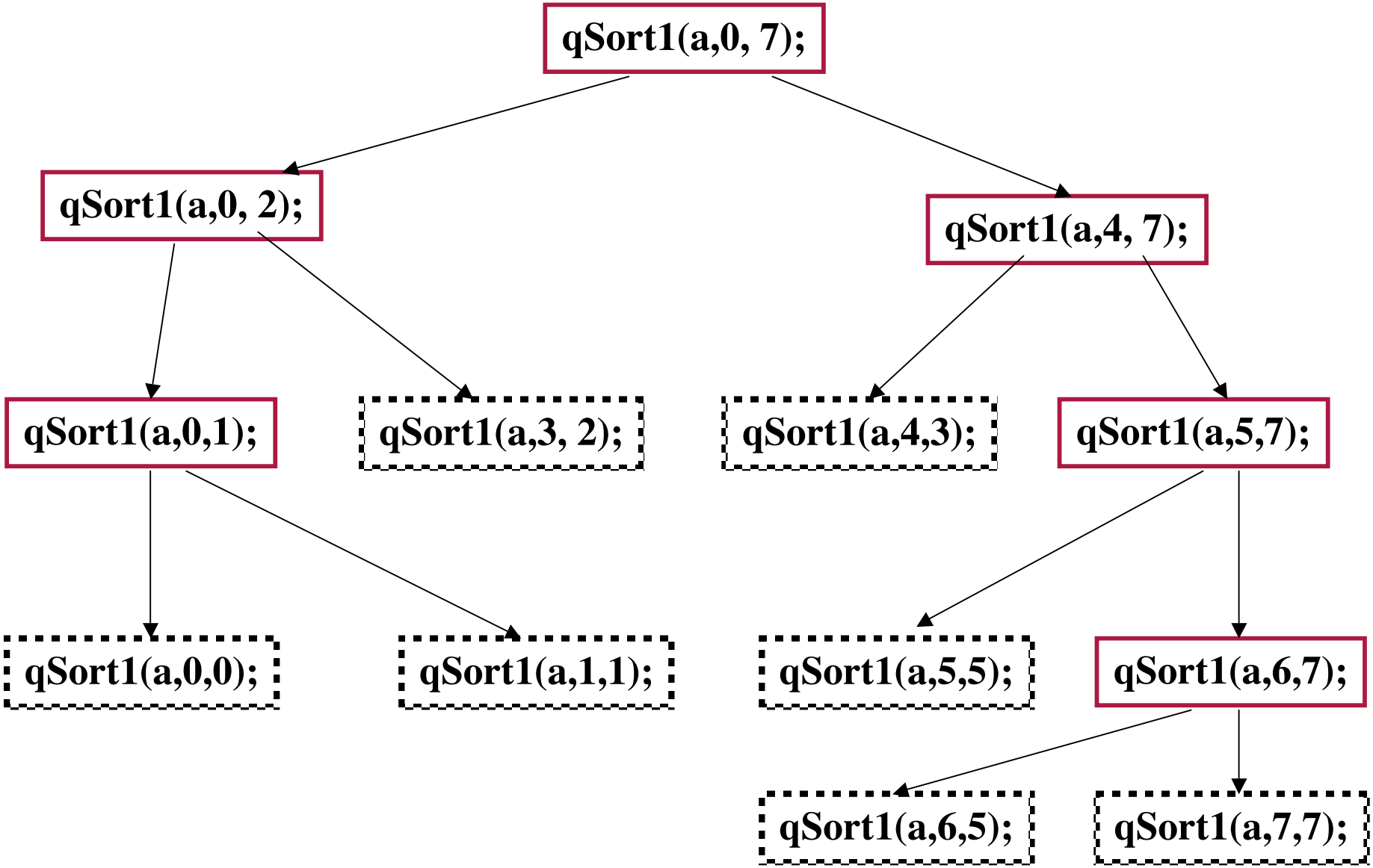





```
m = partition(a,start,end); /*in m il valore di j*/
/*Qui (a[start],…,a[m-1] < a[m]<= a[m+1],…,a[end]*/
    qSort1(a,start, m-1);
    qSort1(a,m+1,end);
```

...

Quicksort - albero delle chiamate



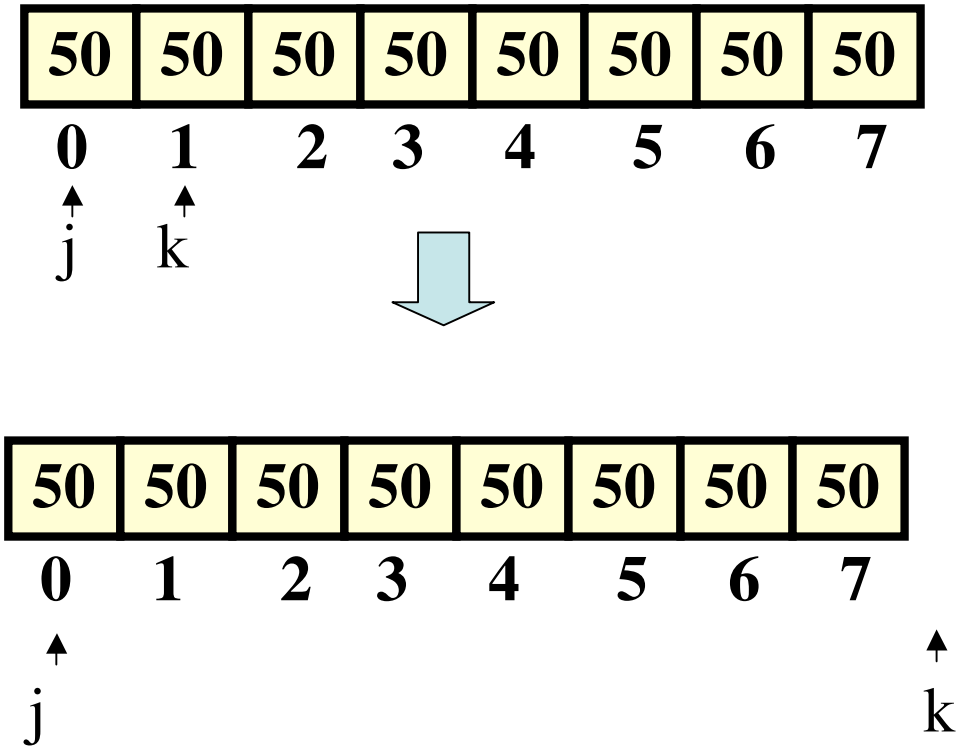
Supponiamo il numero degli elementi n sia pari a 2^h e che si ottenga un albero delle chiamate di altezza h.
In ogni divisione supponiamo che i due sottovettori hanno circa lo stesso numero di elementi
Allora

al livello 0 num. chiamate = 1, ciascuna di costo circa 2^h, tot. $\leq c_0 2^h$
al livello 1 num. chiamate = 2, ciascuna di costo circa 2^{h-1},tot. $\leq c_1 2^h$
al livello 2 num. chiamate = 4, ciascuna di costo circa 2^{h-2},tot. $\leq c_2 2^h$
.
.
.
al livello h num. chiamate $\leq 2^h$, ciascuna di costo costante, tot. $\leq c_h 2^h$

Sommando, il costo è maggiorabile da $ch2^h$, per un'opportuna costante c, cioè costo <= $cn\log_2(n)$

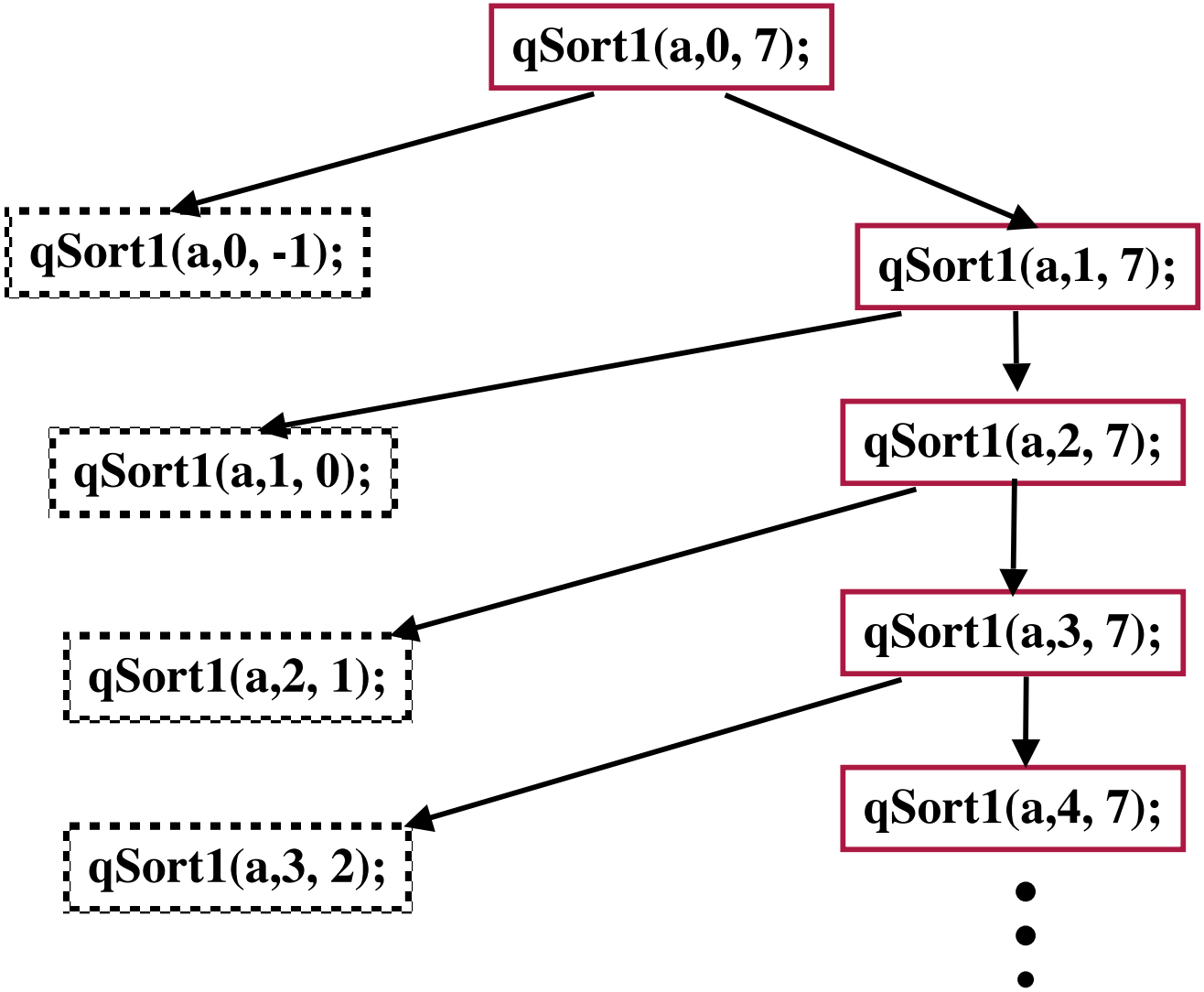
Quicksort - complessità: il caso di elementi tutti uguali

```
{ int j = in, k;
for (k = in+1; k <= fin; k++)
/* invariante:
(a[in+1],...,a[j] < a[in]) &&
(a[j+1],...,a[k-1] >= a[in]) */
   if  (a[k] < a[in])
    scambia(&a[++j],&a[k]);
scambia(&a[j],&a[in]);
return j;
}
```




```
qSort1(a,n,0, -1);
qSort1(a,n,1,end);
...
```

Quicksort - complessità: l'albero delle chiamate nel caso di elementi tutti uguali.



Quicksort - complessità, il caso di n elementi tutti uguali

Abbiamo

al livello 0 num. chiamate = 1, ciascuna di costo circa n, tot. ≤ $c_0 n$
al livello 1 num. chiamate = 2, una trascurabile e una di costo circa n-1,tot. ≤ $c_1(n-1)$
al livello 2 num. chiamate = 2, una trascurabile e una di costo circa n-2,tot. ≤ $c_2(n-2)$.
.
.
al livello n-2 num. chiamate =2 , una trascurabile e una di costo costante d,tot. ≤ $c_{n-1}d$

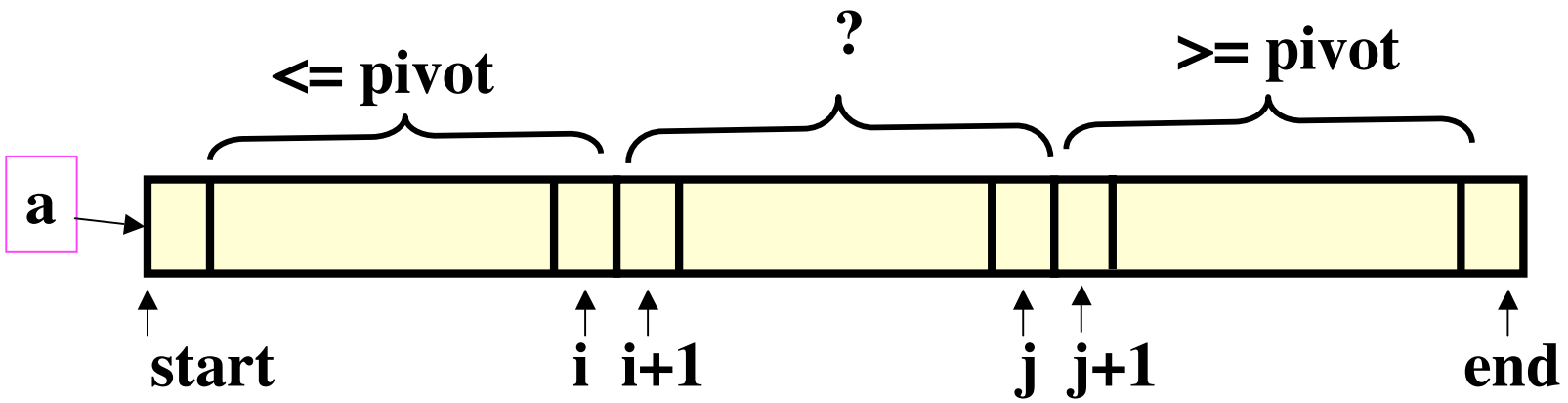
Sommando si ottiene che il costo è maggiorabile da $cn(n+1)/2$, per un'opportuna costante c.

Quicksort - versione alternativa

```
void qSort2(int a[], int start, int end)
{int j;
if (start < end)  /* c'è  più  di un elemento*/
   {j = partition2(a,start,end)
    /*qui (a[start],…,a[j-1] <= a[j]<= a[j+1],…,a[end]*/
    qSort2(a,start, j-1);
    qSort2(a,j+1,end);}
}
```

pivot = a[start]

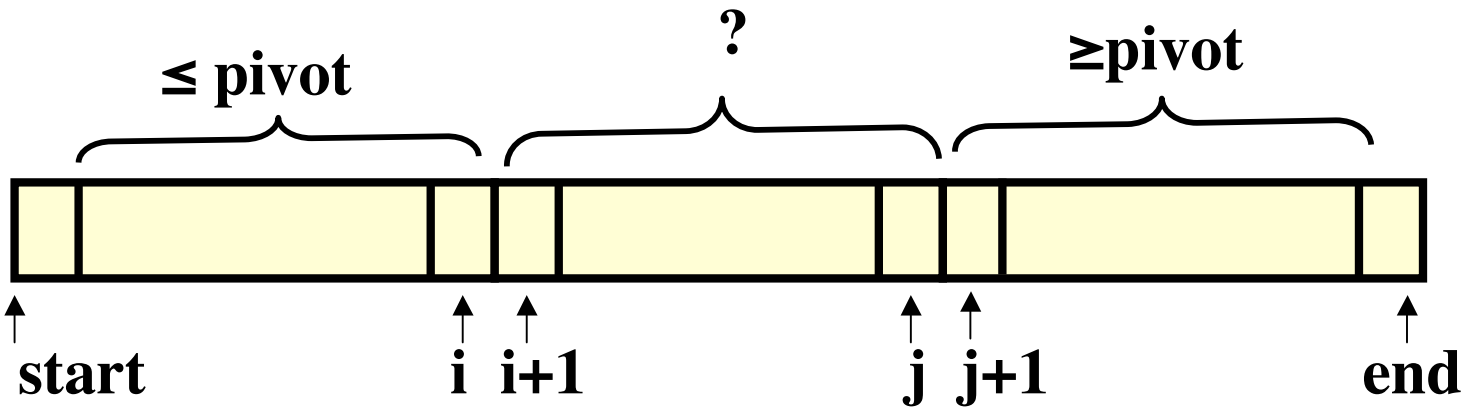
Come migliorare il partizionamento?
Non possiamo pensare di ridurre il numero di confronti ma potremmo puntare a ridurre gli scambi
seconda idea:
scorro l'array da sinistra verso destra finchè trovo un elemento maggiore del pivot, scorro il vettore da destra finchè trovo un elemento più piccolo del pivot, ora scambio questi due e riprendo lo scorrimento.



a[i+1] e a[j] sono gli elementi da confrontare con il pivot e "sistemare" in conseguenza

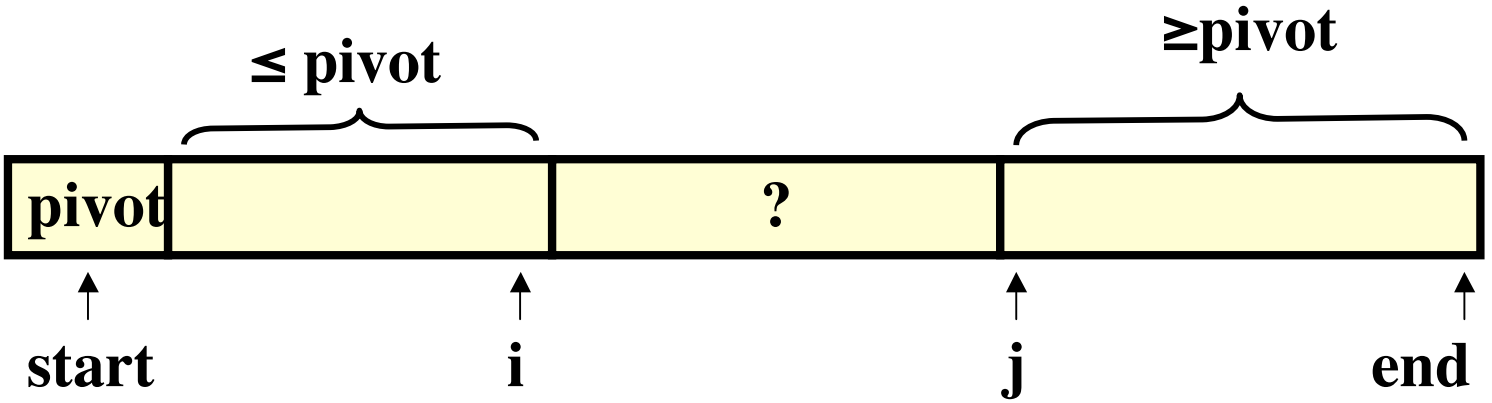
pivot = a[start]

INVARIANTE del ciclo di partizionamento:
(a[start+1],…,a[i] <= pivot)&&
(a[j+1],…,a[end] >= pivot)



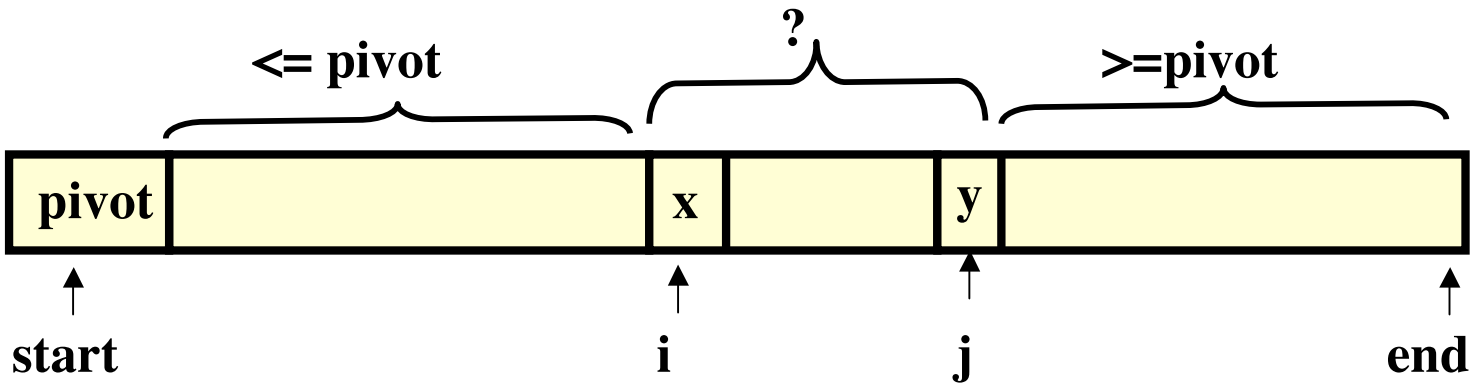
a[i+1] e a[j] sono gli elementi da confrontare con il pivot e "sistemare" in conseguenza

INVARIANTE del ciclo di partizionamento:
(a[start+1],…,a[i] <= pivot)&&(a[j+1],…,a[end] >= pivot)



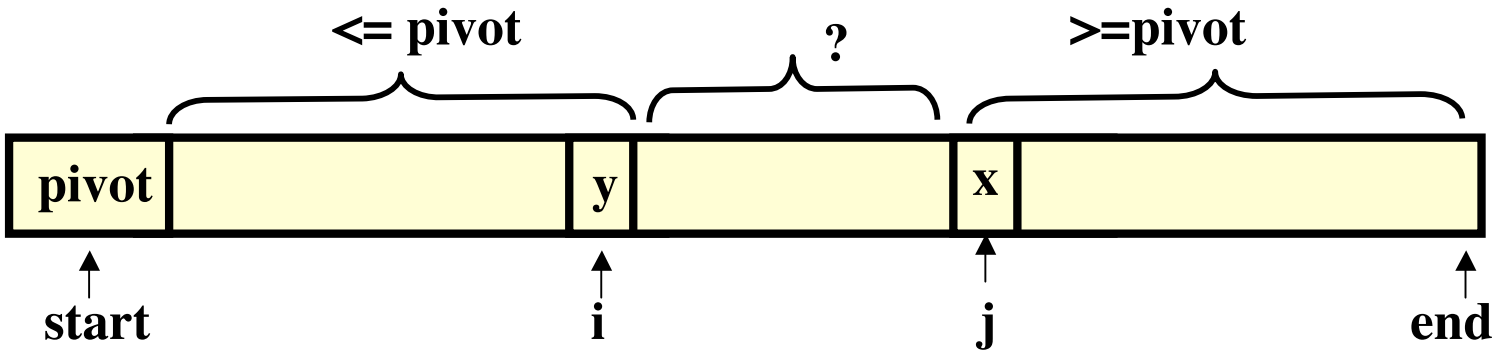

```
i =start+1, j = end;
fintanto che a[i] < pivot incrementa i
fintanto che a[j] > pivot decrementa j
```

Quicksort - partizione2



alla fine dei due cicli
a[i] = x >= pivot e
a[j] = y <= pivot

se effettuo uno scambio:

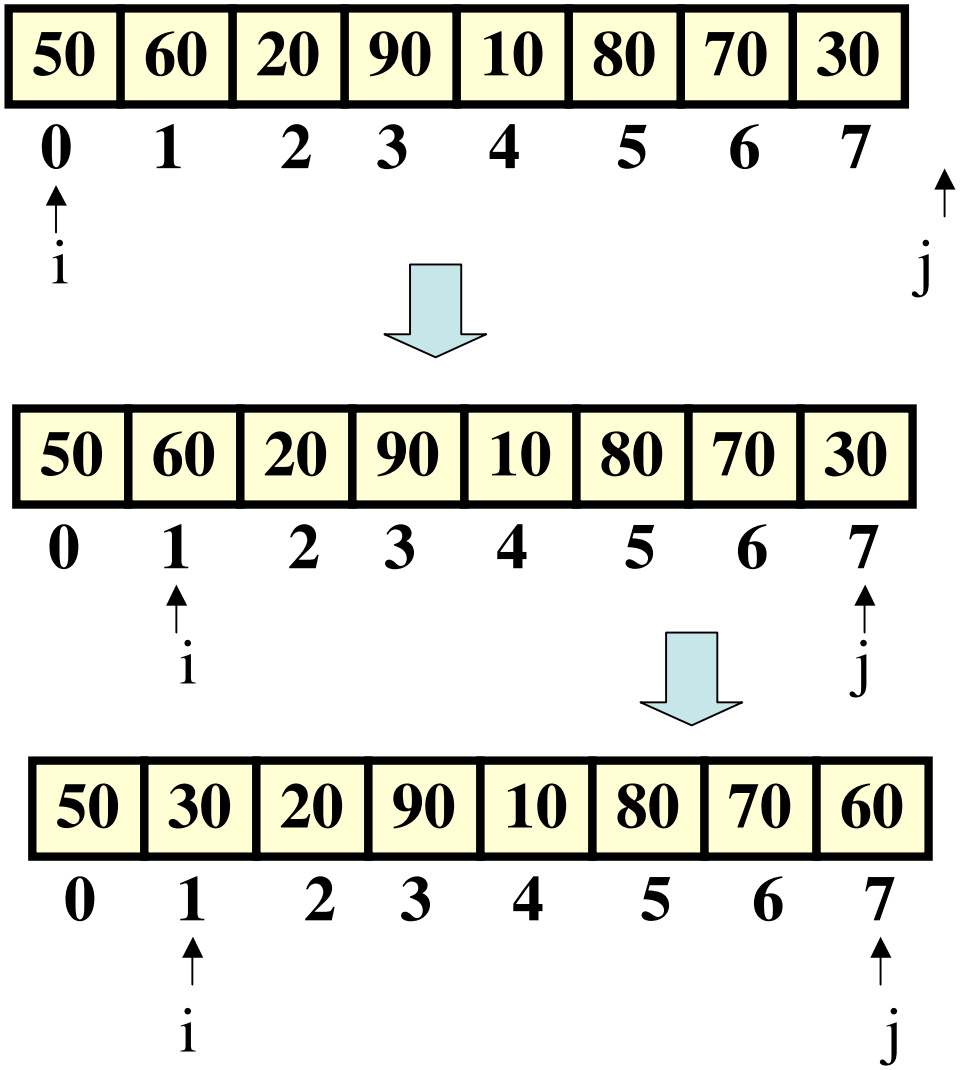


Quicksort - partizione2: il codice

```
int partition2(int a[], int start, int end)
{int pivot =  a[start], i =start, j = end+1;
  for  (;;)
       { /* invariante: (a[start+1],…,a[i] < =pivot)&&
        (a[j+1],…,a[end] >= pivot)*/
        while ((a[++i] < pivot)&&(i<=end));
        while (a[--j] > pivot;
        if  (i > j) break;
        scambia(&a[j],&a[i]);
       }
    scambia(&a[start],&a[j]);
    /*(a[start],…,a[j-1] <= a[j]<= a[j+1],…,a[end]*/
return j;}
```

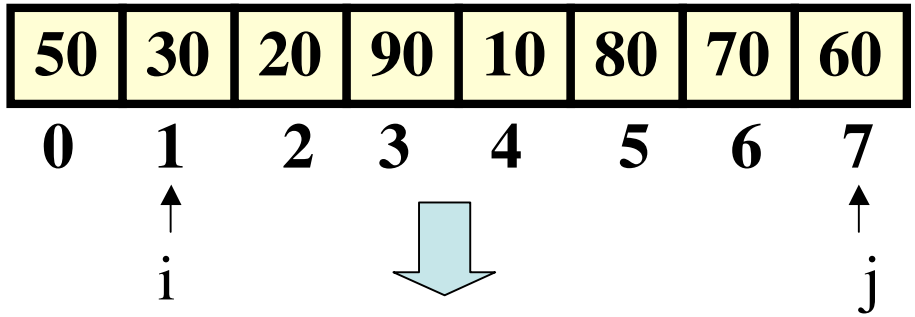
Quicksort - partizione2: esempio di esecuzione

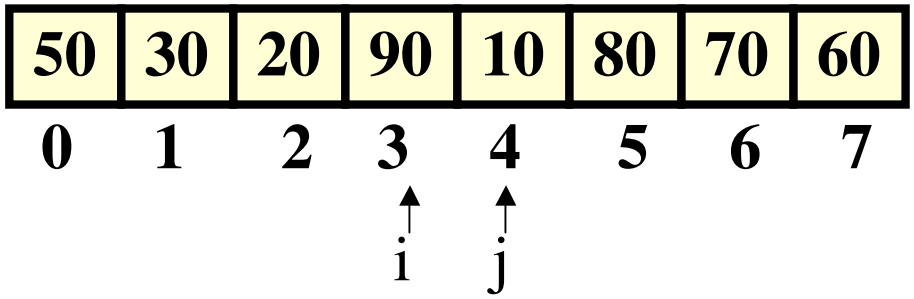
```
{pivot = a[start], i =start,
j = end+1;
for  (;;)
     /* invariante:
(a[start+1],…,a[i] <=pivot)
&&
(a[j+1],…,a[end] >= pivot)*/
  while ((a[++i] < pivot)
          &&   (i<=end))    ;
  while (a[--j] > pivot    ;
 if  (i > j) break;
 scambio(&a[j],&a[i]);
}
```

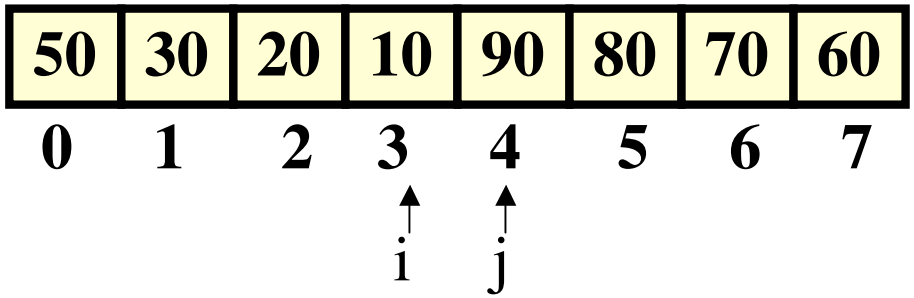


Quicksort - partizione2: esempio di esecuzione

```
{pivot = a[start], i =start,
j = end+1;
for  (;;)
    /* invariante:
(a[start+1],…,a[i] <=pivot)
&&
(a[j+1],…,a[end] >= pivot)*/
  while ((a[++i] < pivot)
          &&   (i<=end))    ;
  while (a[--j] > pivot    ;
 if  (i > j) break;
 scambio(&a[j],&a[i]);
}
```

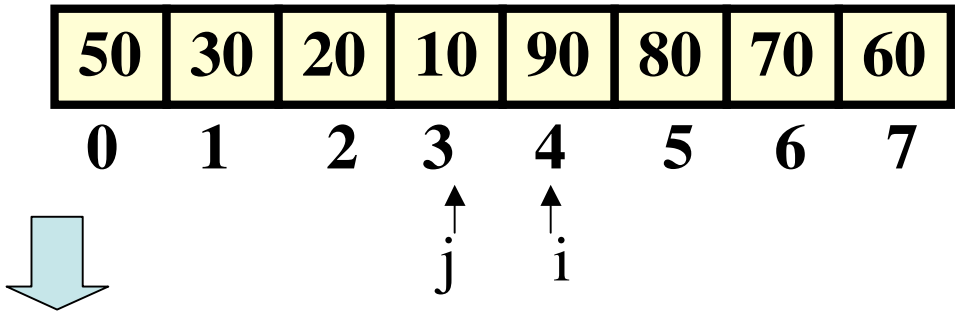




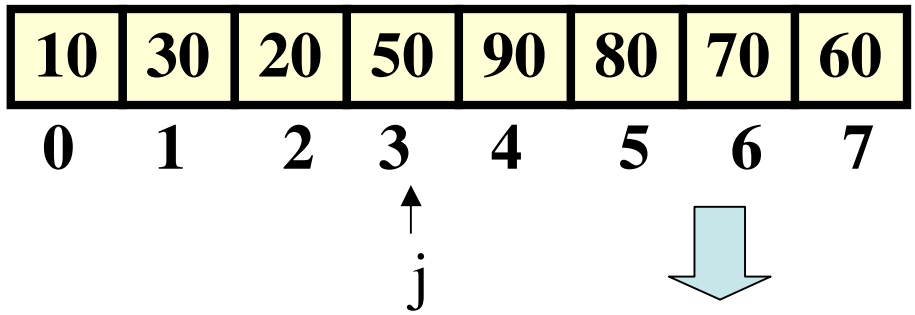


Quicksort - partizione2: esempio di esecuzione

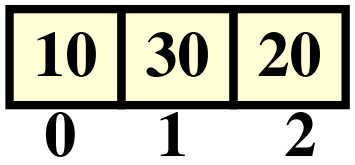
```
if  (i > j) break;
```




```
scambia(&a[start],&a[j]);
```




```
qSort2(a,start, j-1);
```




```
qSort2(a,j+1,end);
```