Ordinamento e Ricerca

Algoritmi di ricerca

Ricerca sequenziale

Si **scorre** l'array elemento per elemento effettuando un **confronto** con l'elemento x che si vuole cercare. Il ciclo termina nel momento in cui l'elemento viene trovato o quando finisce il vettore.

```
int linear_search (TInfo a[], int n, TInfo x) {
    int i =0;
    while (i<n && !equal (a[i], x))
        i ++;
    if (i<n)
        return i;
    else
        return NOT_FOUND ;
}
```

- $T_{best}(n) = \Theta(1)$
- $T_{worst}(n) = T_{avg}(n) = \Theta(n)$

Ricerca dicotomica

Si applica esclusivamente a **vettori ordinati** e si basa su un principio di divisioni successive. Si considera il valore in **posizione centrale** del vettore $chosen = (first + last)/2$ e lo si confronta con il valore x che si vuole cercare.

Se $x > chosen$ si considera il **subarray destro** distinto dagli indici: $fist = chosen + 1$ e $last$; altrimenti si considera il **subarray di sinistra** che ha indici: $first$ e $last = chosen - 1$.

```
int binary_search ( TInfo a[], int n, TInfo x) {
int first = 0;
int last = n-1;
int chosen =(first + last)/2;
    while (first <= last ) {
```

```
        if (equal (a [chosen ], x))
            return chosen ;
        else if (less (a [chosen ], x))
            first = chosen +1;
        else
            last = chosen -1;
        chosen = (first + last )/2;
    }
return NOT_FOUND ;
}
```

- $chosen = (first + last)/2$: valore in posizione centrale del vettore
- $first$: primo elemento utile (inizio del subarray)
- $last$: ultimo elemento utile (fine del subarray)
- $T_{best}(n) = \Theta(1)$
- $T_{worst}(n) = \Theta(\log_2 n)$

Algoritmi di ordinamento

Selection Sort

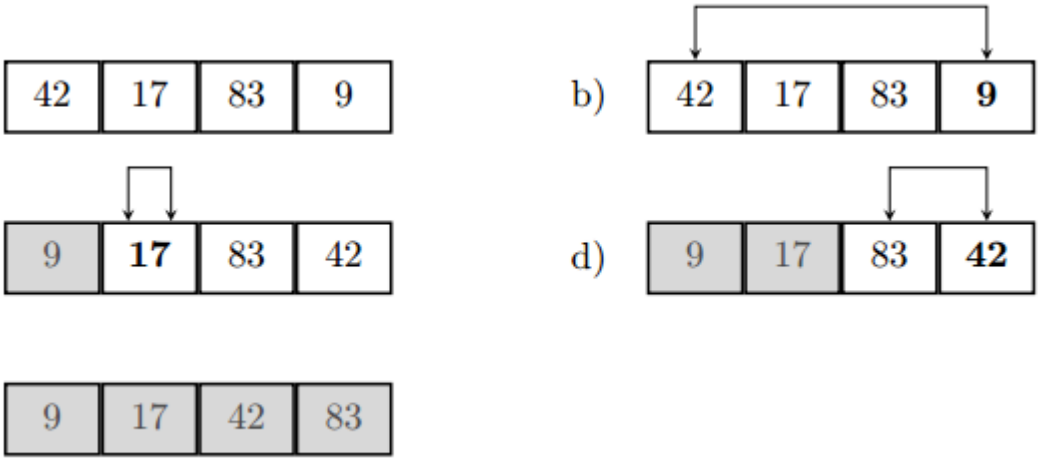
Del vettore di partenza distinguiamo (logicamente) un sottovettore contente gli elementi gia' ordinati ed uno contenente gli elementi ancora da ordinare. L'algoritmo ricerca ciclicamente il **minimo locale** nel sottovettore degli elementi da ordinare e lo posiziona nella prima locazione possibile del sottovettore degli elementi ordinati. E' anche detto "algoritmo per minimi successivi".

```
void selection_sort ( TInfo a[], int n) {
    int i, imin ;
    for (i = 0; i < n-1 ; i ++) {
        imin = i + search_min (a+i, n-i);
        if ( imin != i)
            swap (&a[i], &a[ imin ]);
    }
}

void swap ( TInfo *a, TInfo *b) {
    TInfo temp =*a;
```

```
    *a=*b;
    *b= temp ;
}

int search_min ( TInfo a[], int n) {
    int i, imin ;
    imin = 0;
    for (i = 1; i<n; i ++)
        if ( less (a[i], a [imin]))
            imin =i;
    return imin ;
}
```

- $search\_min$: ricerca il minimo nel vettore passato come parametro. Restituisce un indice (posizione) **relativo** dato che la funziona ha lavorato su un sottoarray di a. Combinandolo con i $(i + search\_min)$ si ottiene l'indice **assoluto**.
  - $a + i$: riferimento al sottovettore da ordinare
  - $n - i$: elementi da ordinare
  - i : elementi gia' ordinati
- $swap$: scambia l'elemento in posizione i con l'elemento presente all'indice $imin$ precedentemente cercato.

$T(n) = \Theta(n^2)$



selection sort

▼ Insertion Sort

Si basa sul principio dell'i**nserimento in ordine** (anche detto "algoritmo per inserimenti successivi") in un vettore gia' ordinato, tuttavia e' presente (e qui considerata) la versione in-place ottenuta andando a **simulare** l'inserimento di un valore in un vettore vuoto o gia' ordinato. I

I principio fondamentale e' l'individuazione dell'**indice di inserimento** del valore x e la creazione di uno **spazio** per l'inserimento di tale valore.

Anche in questo caso distinguiamo un sottoarray di elementi gia' ordinati (le prime posizioni dell'array) ed uno di elementi da ordinare (le ultime posizioni).

```
/* Ordinamento per inserimenti successivi */
void insertion_sort (TInfo a[], int len) {
   int i;
   for (i=1; i<len; i ++)
      insert_in_order (a, i, a[i]);
}

// Inserisce un elemento in un array gia ' ordinato
// in modo da mantenere l' ordinamento .

void insert_in_order (TInfo a[], int n, TInfo x) {
   int pos , i;

/* Cerca la posizione di inserimento */
```

```
    for (pos = n; pos >0 && greater(a[pos-1] , x); pos --);

/* Sposta in avanti gli elementi successivi */
    for (i= n-1; i >= pos ; i --)
        a[i+1] = a[i];

/* Inserisce l' elemento */
    a[pos] = x;
}
```

- $insertion\_sort$: **scorre (partire dalla 2 posizione) tutto il vettore** a. Serve sostanzialmente a dividere logicamente il vettore nei 2 subarray. i rappresenta infatti l'indice del primo elemento da ordinare... $len - i$ rappresenta pertanto il numero di elementi gia' ordinati.

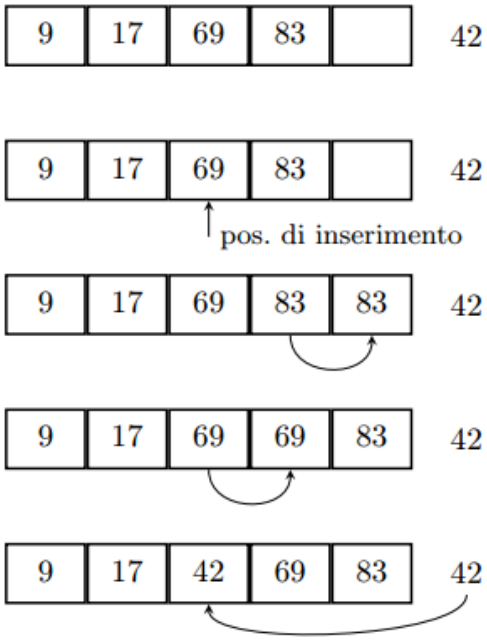
  $T_{best}(n) = \Theta(n)$

  $T_{worst}(n) = T_{avg}(n) = \Theta(n^2)$

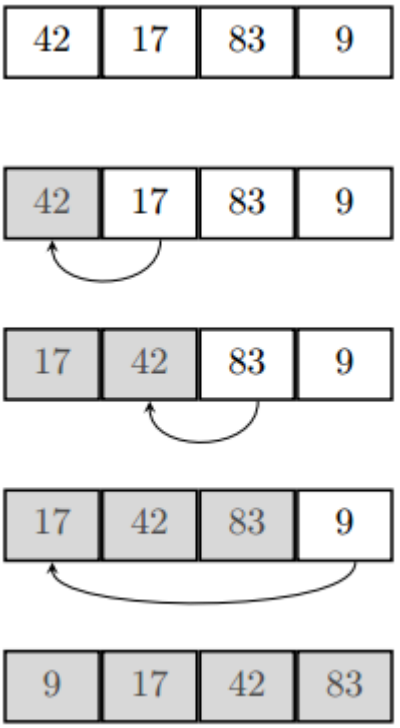
- $insert\_in\_order$: inserisce un elemento in un array gia' ordinato. L'inserimento avviene in 3 step:

  Si cerca l'**indice di inserimento**, si crea uno spazio per l'inserimento del valore spostando tutti gli elementi $i...n$ a destra, si inserisce l'elemento.

  - a : riferimento del vettore nella sua interezza. Si passa il vettore per intero ma in realta' si considerano solamente gli elementi da ordinare (ovvero il subarray di destra);
  - $n = i$: indice iniziale del sottoarray degli elementi da ordinare;
  - $x = a[i]$: **elemento** da inserire. Il passaggio per valore viene sfruttato per copiare l'elemento da inserire il quale altrimenti verrebbe sovrascritto durante la fase di shift.



insert_in_order



insertion sort

▼ Bubble Sort

L'algoritmo si basa su una **serie di confronti** degli elementi adiacenti dell'array a : $a[i]$, $a[i+1]$. Se $a[i] > a[i+1]$ (il precedente e' maggiore del successivo), i due numeri vengono scambiati usando la funzione di swap.

Il principio di funzionamento consiste nello **spostare i numeri piu pesanti**, quelli piu grandi, alla fine dell'array. Ogni scansione dell'array posiziona in coda l'elemento piu grande presente.

Il primo for gestisce il numero di scansioni da effettuare, il for interno invece serve ad esaminare tutte le **coppie** di valori valutando se effettuare o meno lo scambio.

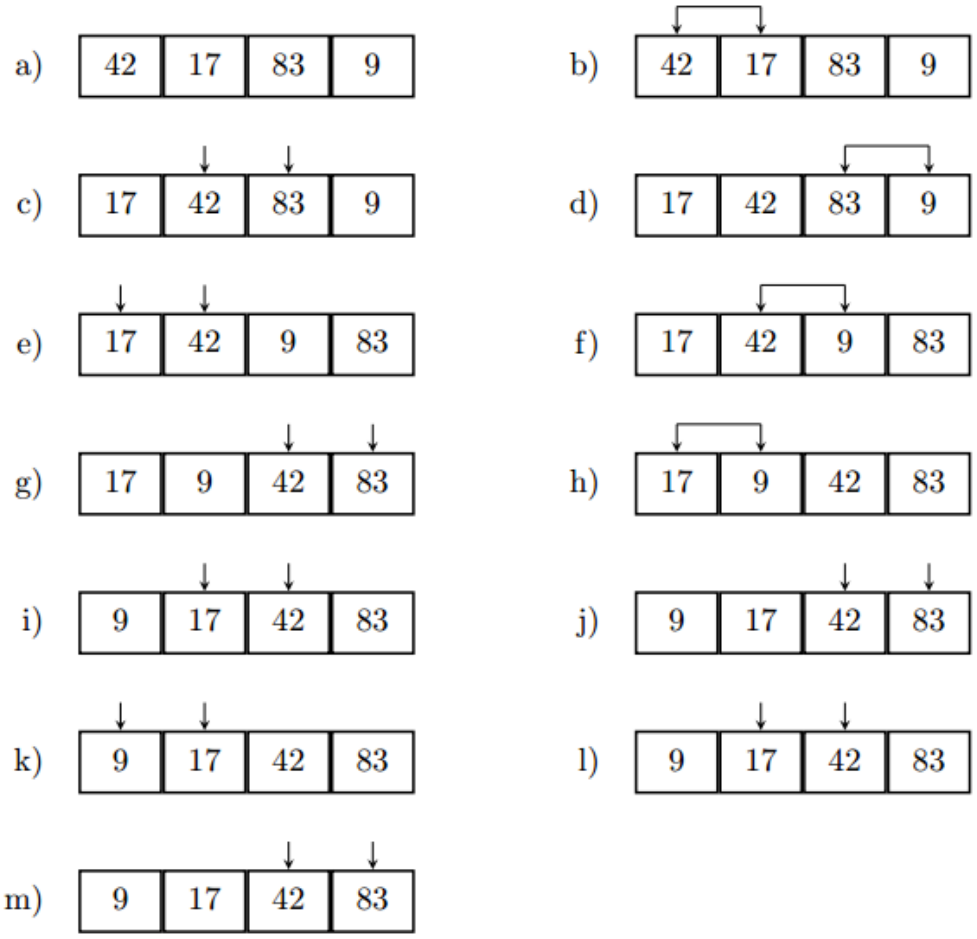
```
void bubble_sort ( TInfo a[], int n) {
   int i, k;
   bool modified ;
   modified = true ;

   //gestisce il numero di scansioni
```

```
    for (k = 0; k < n-1 && modified ; k ++) {
        modified = false ;
        //esamina tutte le coppie di valori
        for (i = 0; i < n-k-1; i++)
            //se il prec>succ scambia i valori
            if ( greater (a[i], a[i+1])) {
                swap (&a[i], &a[i +1]);
                modified = true ;
            }
    }
}

void swap ( TInfo *a, TInfo *b) {
    TInfo temp =*a;
    *a=*b;
    *b= temp ;
}
```

- $modified$: **sentinella**, indica se durante l'iterazione precedente e' stato scambiato qualche valore all'interno del vettore. Il valore $false$ indica pertanto che il vettore considerato e' **gia' ordinato** e, per questo motivo, viene utilizzata come condizione per primo ciclo for.
- k : scansioni dell'array effettuate
- $n-k-i$: Rappresenta la **fine del subarray da ordinare**. Il subarray di destra e' gia' ordinato dato che ogni iterazione colloca in coda l'elemento piu' grande presente all'interno dell'intero array... e' pertanto inutile considerare nuovamente tali elementi.

  $T_{best}(n) = \Theta(n)$

  $T_{best}(n) = \Theta(n^2)$



bubble sort

▼ Merge Sort

L'algoritmo si basa sulla **fusione di array gia' ordinati** (compito della funzione $merge$). La funzione $merge\_sort$ ha il compito di dividere l'array a in 2 subarray a', a'' di dimensioni $m = \frac{n}{2}$, $n - m$ e **riordinarli** ricorsivamente. L'invocazione della funzione $merge$ consente di fondere i due sottoarray a', a" in un unico array c.

Sostanzialmente **da un vettore di dimensione n vengono prodotti n vettori di dimensione 1** (per definizione ordinati) i quali vengono successivamente fusi insieme man mano dalla funzione $merge$.

```
//divide in 2 sub e li riordina ricorsivamente
void merge_sort (TInfo a[], int n, TInfo temp []) {
   int i, m=n /2;
   if (n <2)
      return ;
   merge_sort (a, m, temp );      //riordina sub-left
```

```
    merge_sort (a+m, n-m, temp );   //riordina sub-right
    merge (a, m, a+m, n-m, temp );  //fondi i 2 sub
    for (i =0; i<n; i ++)
        a[i]= temp [i];             //copia nel vett originale
}

//fonde due array ordinati in un unico array
void merge ( TInfo a1 [], int n1 , TInfo a2 [], int n2 ,TInfo dest []) {
    int pos1 = 0, pos2 = 0, k = 0;
    while (pos1 < n1 && pos2 < n2) {
        if (less (a2[ pos2 ], a1[ pos1 ]))
            dest [k ++] = a2[pos2 ++];
        else
            dest [k ++] = a1[pos1 ++];
    }
    while (pos1 <n1)
        dest [k ++] = a1[pos1 ++];
    while (pos2 <n2)
        dest [k ++] = a2[pos2 ++];
}
```

- $merge\_sort$: **divide** l'array di partenza a in 2 sottoarray: a_1 di dimensione $m = \frac{n}{2}$ ed indici $0, m-1$; a_2 di dimensione $n-m$ ed indici $m, n-1$.

  La **prima chiamata ricorsiva** ordina il subarray a_1, la **seconda chiamata** invece ordina a_2. A questo punto $merge$ fonde i due sottoarray a_1, a_2 gia' ordinati in un unico array. Il for serve a copiare nell'array di partenza i valori contenuti nel vettore temporaneo $temp$;

  - m : dimensione di a_1;
  - $temp[]$: array di appoggio, l'algoritmo non lavora in-place;
  - $a + m$: riferimento del **subarray di destra**, a_2 ovvero l'indirizzo dell'elemento $a[m]$ dove m rappresenta la dimensione di a_1;
  - $n - m$: dimensione di a_2\

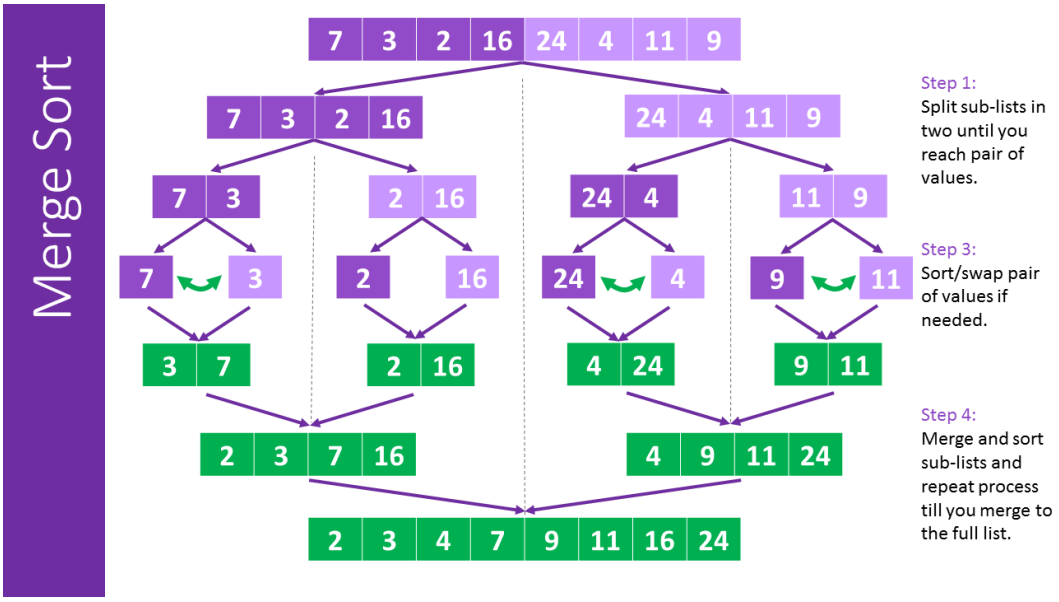
  $T(n) = \Theta(n \log n)$

- $merge$: esegue la **fusione** di due array gia' ordinati in un unico array. Il **primo** $while$ serve a **confrontare i primi elementi** non utilizzati di a_1 ed a_2: $a_1[pos_1] < a_2[pos_2]$, il **minore** tra i due viene copiato nell'array di destinazione $dest$.

  I $while$ **successivi** servono per copiare gli elementi rimanenti (non utilizzati quindi) dei vettori a_1 ed a_2 in $dest$.

  - a_1 : subarray sinistro;
  - $n_1 = m$: dimensione di a_1
  - a_2 : subarray destro;
  - $n_2 = n - m$: dimensione di a_2;
  - $pos*$: indice del primo elemento da considerare di $a1$ ed a_2, gli elementi precedenti sono gia' stati considerati e pertanto inseriti in $dest$;
  - k : elementi gia' copiati nell'array di destinazione;

  $T_{merge}(n) = \Theta(n)$



▼ Quick Sort

Effettua l'ordinamento sul posto e si basa sulla divisione dell'array in **3 subarray** detti partizioni (funzione $partition$).

```
/*Ordina ricorsivamente le partizioni dell'array create dalla partition */
void quick_sort ( TInfo a[], int n) {
   int k;
   if (n <2)
      return ;
   k = partition (a,n);
   quick_sort (a,k);            //ordina subarray sinistro
   quick_sort (a+k+1, n-k-1);    //ordina subarray destro
}

/*divide l'array in 3 parti:
  - 1: elementi < del pivot
  - 2: pivot
  - 3: elementi >= del pivot

   Restituisce l'indice del pivot
*/

int partition ( TInfo a[], int n) {
int i, k = 1;
for (i=1; i<n; i ++)

   //se il valore e' < pivot swap con l'ultimo elemento del sub-left
   if ( less (a[i], a [0]))
      swap (&a[i], &a[k++]);
swap (&a[0] , &a[k-1]);      //swap pivot con ultimo elemento sub-left
return k-1;
}

void swap ( TInfo *a, TInfo *b) {
   TInfo temp =*a;
   *a=*b;
   *b= temp ;
}
```

- $quick\_sort$: effettua l'ordinamento dei 3 subarray generati da $partition$ e li **combina** in un unico vettore. La prima chiamata

ricorsiva ordina il sottovettore di sinistra, la seconda chiamata quello di destra;

- k : indice del pivot calcolato dalla $partition$ ($k-1$ rappresenta il numero di elementi presenti nel subarray sinistro);
- $a+k+1$: riferimento al sottoarray di destra contenente i valori maggiori del pivot (ovvero l'indirizzo dell'elemento immediatamente successivo al pivot);
- $n-k-1$: dimensione sottoarray di destra (numero elementi maggiori del pivot);

$T_{best}(n) = T_{avg}(n) = \Theta(n \log n)$

$T_{worst}(n) = \Theta(n^2)$

Il caso peggiore si verifica quando il pivot e' il massimo o il minimo dell'array da partizionare. Paradossalmente il $Quick\ Sort$ e' inefficiente se il vettore e' gia' ordinato.

- $partition$: effettua la **divisione** dell'array a in 3 sottoarray contenenti rispettivamente:
  1. a' : subarray sinistro, contiene gli elementi **minori** del pivot;
  2. $a"$: subarray centrale, contiene solamente il **pivot**;
  3. a''' :subarray destro, contiene gli elementi **maggiori** del pivot

  E restituisce l'**indice del pivot** a seguito dell'inserimento nel sottoarray centrale.

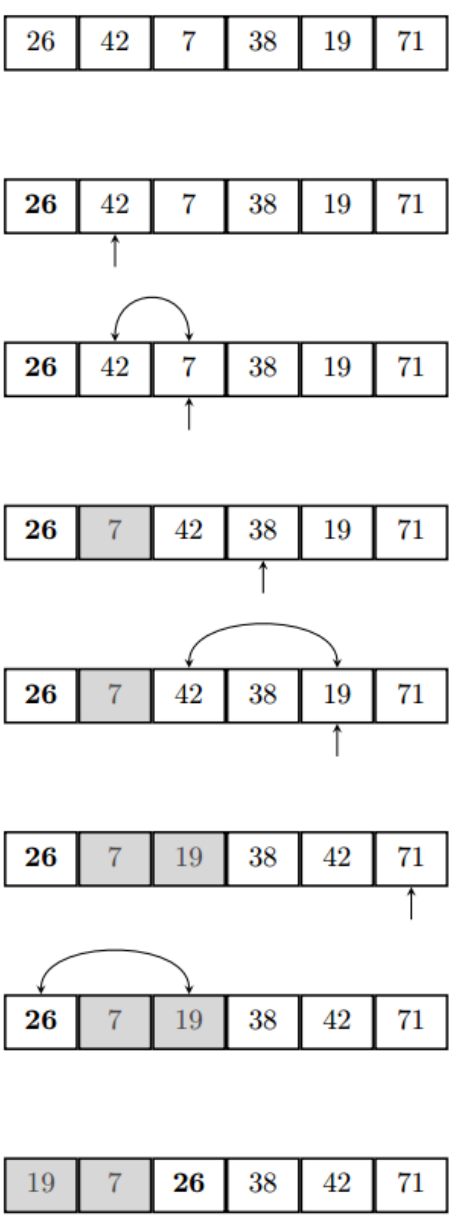
  Il pivot inizialmente corrisponde al primo elemento del vettore a ovvero $a[0]$. Viene posizionato nel relativo sottoarray solamente prima della conclusione delle funzione $partition$.

  Il for serve a **scorrere gli elementi a destra del pivot** (il quale si trova in posizione $a[0]$) i quali vengono man mano **confrontati con il pivot**... se l'elemento e' **minore** del pivot ($a[i] < a[0]$) si effettua lo **scambio** con **l'ultimo elemento del sottoarray di sinistra** (viene quindi spostato in coda alla parte iniziale dell'array, scambiandolo con l'elemento di indice k : $swap(\&a[i], \&a[k++])$.

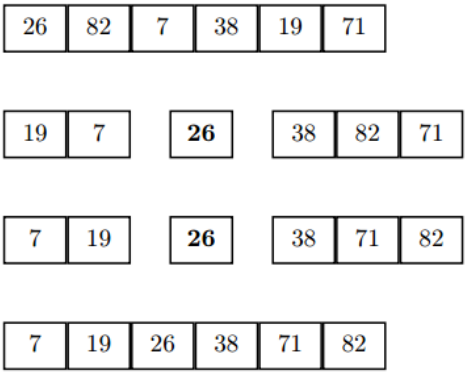
  Alla fine del for (quando si termina lo scorrimento dell'intero vettore), si posiziona il **pivot nella posizione corretta** (ovvero nel suo

sottoarray) scambiandolo con l'ultimo elemento del subarray di sinistra, di indice $k-1$: $swap(\&a[0], \&a[k-1])$.

$T_{partition}(n) = \Theta(n)$



Partition



Quicksort