Algoritmi di ordinamento

- Obiettivi:
  - Presentare alcuni algoritmi di ordinamento di vettori (array in memoria centrale) e discuterne la complessità in tempo
  - Presentare la funzione **`qsort`** della libreria **`stdlib.h`** e il suo utilizzo

Ordinamento

Dato un insieme S di n oggetti presi da un dominio totalmente ordinato, ordinare S

- Esempi: ordinare una lista di nomi alfabeticamente, o un insieme di numeri, o un insieme di compiti d'esame in base al cognome dello studente
- Subroutine di molti problemi
- Abbiamo visto che è possibile effettuare ricerche in array ordinati in tempo $O(\log_2 n)$

Algoritmi di Ordinamento

- Occorre avere a che fare con array di elementi sul cui tipo sia definita una ***relazione d'ordine totale***
  - Ad esempio, tipo **`float`**
  - Più in generale sui tipi scalari

- Per semplicità si suppongano effettuate le dichiarazioni che seguono:

```
#define N 11
typedef float ELEMENT;
typedef ELEMENT ARRAY[N];
typedef enum {false, true} Boolean;
```

Algoritmi di Ordinamento

- Si supponga definita la funzione che segue

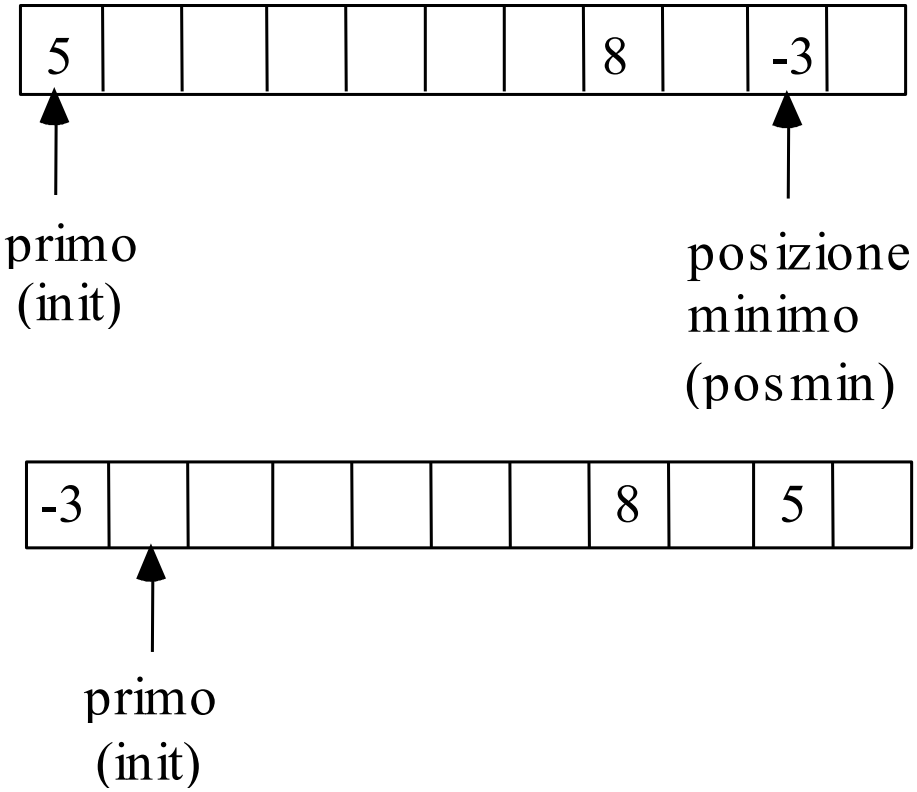
```
void swap(ELEMENT *a, ELEMENT *b)
{
    ELEMENT tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Ordinamenti quadratici – $O(N^2)$

- Naive sort (o selection sort)
- Bubble sort

Naive Sort

- Detto anche Selection Sort o ordinamento per minimi successivi
- Ad ogni passo seleziona il minimo nel vettore e lo pone nella prima posizione, richiamandosi ed escludendo dal vettore il primo elemento



Naive Sort

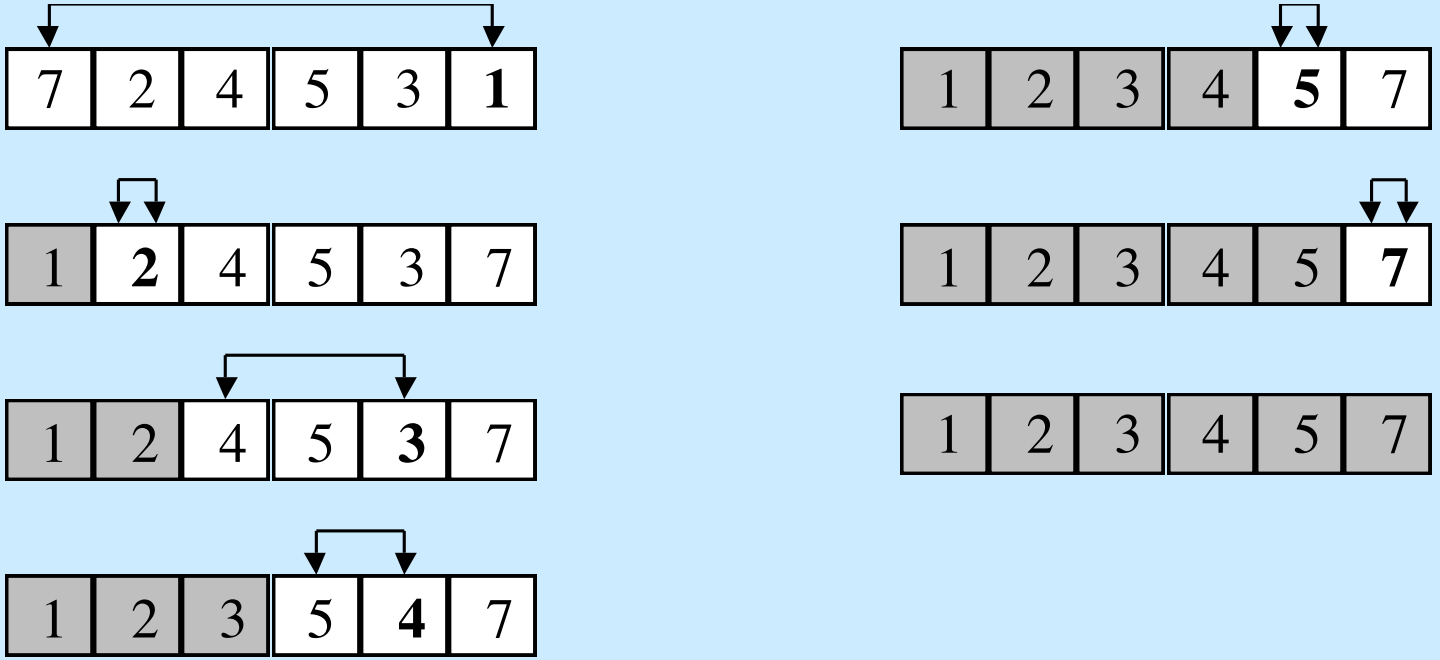
- Pseudo codifica – versione ricorsiva

<se array corrente ha un solo elemento allora è già ordinato – termina>
<individua il minimo nell'array corrente, di posizione posmin>
<scambia se necessario il primo elemento dell'array
corrente con A[posmin]>
<ordina l'array ottenuto eliminando il primo elemento>

- Pseudo codifica – versione iterativa

```
while (array ha più di un elemento)
{
        <individua il minimo nell'array corrente, di posizione posmin >
        <scambia se necessario il primo elemento dell'array
                corrente con A[posmin]>
        <considera come array corrente quello
                precedente tolto il primo elemento>
}
```

Naive sort: esempio



Naive Sort – Iterativo

```
void naiveSort(ARRAY a, int dim)
{
    int  j, i, posmin;
    ELEMENT min;

    for  (j = 0; j < dim; j++)
    {
        posmin = j;
        for (min = a[j], i = j + 1; i < dim; i++)
            if (a[i] < min)
            {
                posmin = i;
                min = a[i];
            }

        if (posmin != j)
            swap(&a[j], &a[posmin]);
    }
}
```

Naive Sort – Complessità $O(n^2)$

- Array di dimensione n
- La ricerca del minimo si ripete n-1 volte (ciclo **`for`** esterno)
- Per trovare il minimo, l'istruzione di confronto (**`(a[i]<min)`** dominante), è eseguita:

  n-1 n-2 ... 3 2 1 volte (i)

  1 2 ... n-3 n-2 n-1 passo (j+1)

- $\sum_{(i=1..n-1)} i = n*(n-1)/2 =$ **$O(n^2)$**
- Questo risultato è ***indipendente dai valori di ingresso***
- Comunque si eseguono **$O(n^2)$** confronti anche se il vettore è già ordinato

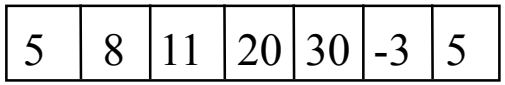
Ordinamenti quadratici – $O(N^2)$

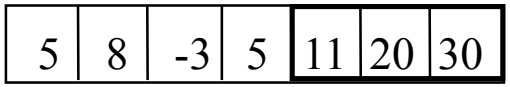
- Naive sort (o selection sort)
- Bubble sort

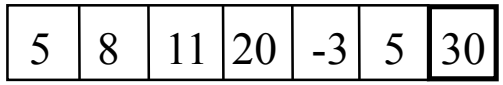
Bubble Sort

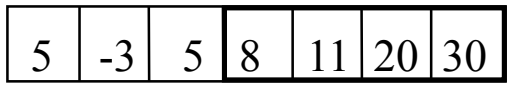
- Ordinamento a ***"bolla"***
- Effettua una serie di (al più) N-1 scansioni dell'array
- Ad ogni scansione, si confrontano le coppie di elementi adiacenti **`a[i]`** e **`a[i+1]`**, che, se non rispettano l'ordinamento, vengono scambiati di posizione tra loro
- Il vettore è ordinato quando non ci sono più scambi

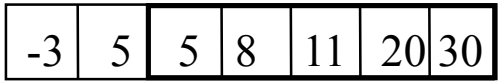
Bubble Sort - esempio











Bubble Sort

- Pseudo codifica – versione ricorsiva

  <per tutte le coppie di elementi adiacenti dell'array
  corrente **a** esegui:>
  <se **a[i] > a[i+1]** allora scambiali>
  <se l'array non è ordinato...>
  <ordina l'array ottenuto da a eliminando l'ultimo elemento>

- Pseudo codifica – versione iterativa

  do
  <per tutte le coppie di elementi adiacenti dell'array
  corrente **a** esegui:>
  <se **a[i] > a[i+1]** allora scambiali>
  <imposta come array corrente l'array **a** tolto l'ultimo elemento>
  while (<il vettore **a** non è ordinato>)

Bubble Sort

- Come riconoscere se l'array è ordinato o meno?
  - Il vettore è ordinato **quando non ci sono più scambi**!
- Si chiama ***ordinamento a bolla*** perché dopo la prima scansione dell'array, l'elemento massimo si porta in ultima posizione (gli elementi più piccoli "salgono" verso le posizioni iniziali del vettore)
- Ogni "passata" ha come effetto la collocazione nella sua posizione definitiva di un elemento:
  - la prima scansione pone il valore massimo in ultima posizione...
  - la seconda colloca il massimo tra gli elementi rimanenti nella penultima posizione...
- Ad ogni scansione è possibile ridurre l'array alla parte non ancora ordinata

Naive Sort – Iterativo

```
void naiveSort(ARRAY a, int dim)
{
    int  j, i, posmin;
    ELEMENT min;

    for  (j = 0; j < dim; j++)
    {
        posmin = j;
        for (min = a[j], i = j + 1; i < dim; i++)
            if (a[i] < min)
            {
                posmin = i;
                min = a[i];
            }

        if (posmin != j)
            swap(&a[j], &a[posmin]);
    }
}
```

Bubble Sort – Iterativo

```
void bubbleSort(ARRAY a, int dim)
{
    Boolean swapped;
    int i;

    do
    {
        swapped = false;
        for (i = 0; i < dim - 1; i++)
        {
            if ( a[i] > a[i + 1] )
            {
                swapped = true;
                swap(&a[i], &a[i + 1]);
            }
        }
        dim--;
    }
    while (swapped);
```

Bubble Sort - Complessità

- ***Dipende dai valori dei dati in ingresso***
- Non sono sempre necessarie n-1 iterazioni esterne
  - se non avviene alcuno scambio, l'algoritmo termina dopo la prima iterazione
- ***Caso migliore:*** array già ordinato

1	2	3	4	7	12	15	18

  - Una sola iterazione esterna, con n-1 confronti e nessuno scambio, O(n)
- ***Caso peggiore:*** array ordinato in senso decrescente

18	15	12	7	4	3	2	1

Bubble Sort – Caso peggiore

- ***Caso peggiore:*** array ordinato in senso decrescente
  - Al passo di iterazione i, (n-i) confronti e (n-i) scambi

1	*2*	*…*	*n-3*	*n-2*	*n-1*	*passo*
n-1	n-2	…	3	2	1	confr
n-1	n-2	…	3	2	1	swap

$\sum_{(i=1..n-1)} i = (n-1)*n/2$

è sempre la serie di Gauss

  - L'ordine di grandezza del numero di scambi è **$O(n^2)$**, in modo analogo a quello di Naive Sort

Complessità Ordinamento

- Il problema di ordinare una sequenza di N elementi ha una delimitazione inferiore di complessità pari a $N*\log_2 N$
- Esistono algoritmi migliori di Naive e Bubble Sort?
- Algoritmi ottimi hanno complessità $O(N*\log_2 N)$ – li vedremo nelle parti successive della lezione.

Ordinamenti ottimi

- Merge sort
- Quick sort

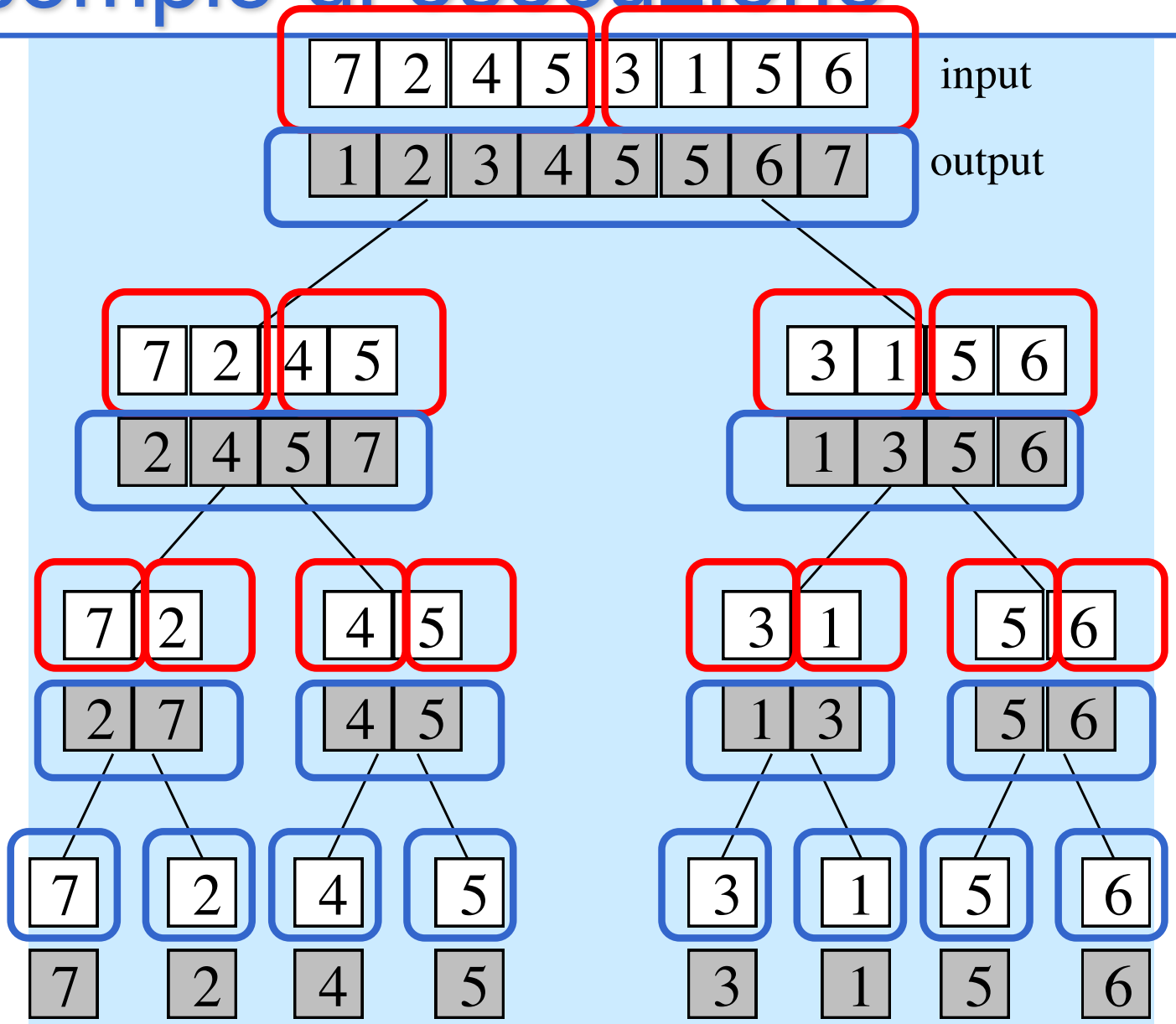
Merge Sort

- Ordinamento ***per fusione***

- "Divide et impera"
  - Divide a metà il vettore
  - Ordina le due metà (ricorsione)
  - Fonde le due metà ordinate

- Utilizza un algoritmo di Merge

Esempio di esecuzione



Merge Sort

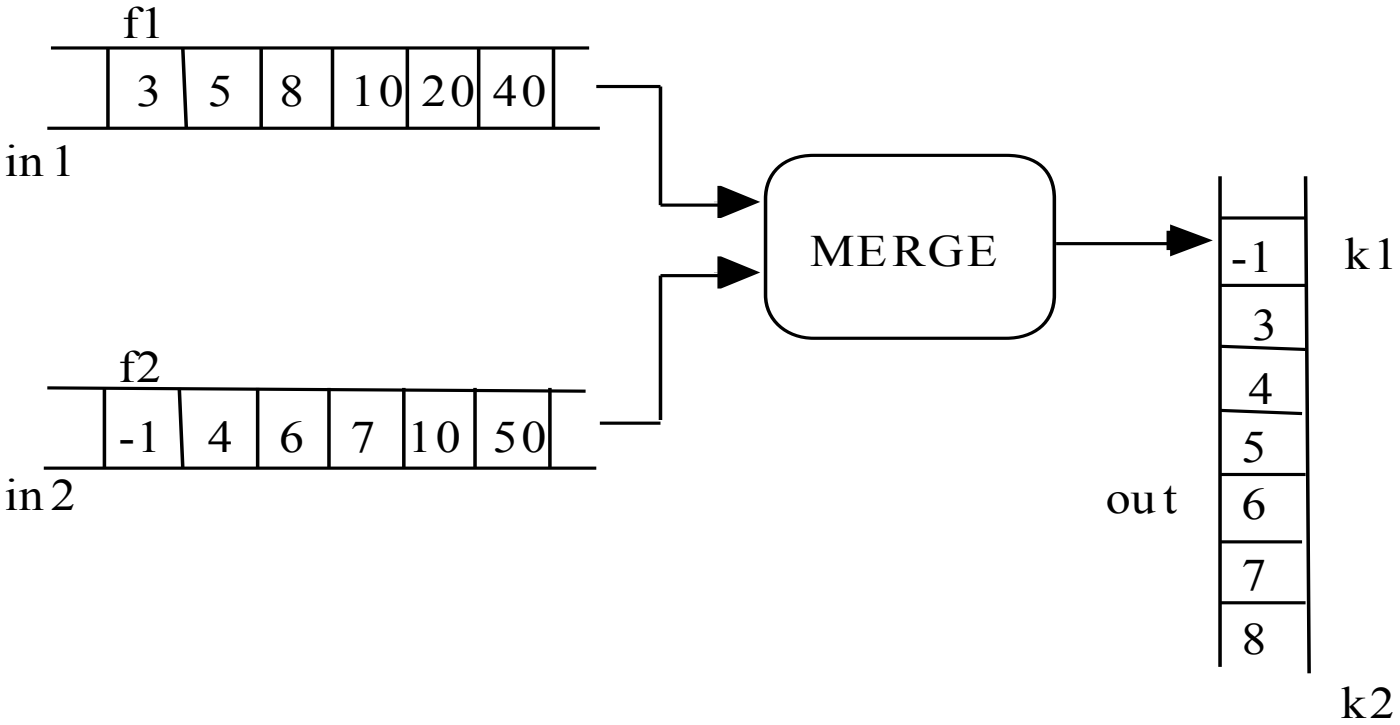
```
void merge(ARRAY a, int iniz1, int iniz2, int fine);
void mergeSortR(ARRAY a, int iniz, int fine);

void mergeSort(ARRAY a, int dim)
{
    mergeSortR(a, 0, dim - 1);
}

void mergeSortR(ARRAY a, int iniz, int fine)
{
    if (iniz < fine)
    {
        int m = (fine + iniz) / 2;
        mergeSortR(a, iniz , m);
        mergeSortR(a, m + 1, fine);
        merge(a, iniz, m + 1, fine);
    }
}
```

…mantiene uniformi le signature delle funzioni di ordinamento…

Algoritmo di Merge

- Dati due vettori **`x`**, **`y`** con **`m`** componenti ciascuno e ordinati in ordine crescente, produce un unico vettore **`z`**, di **`2*m`** componenti ordinato
- Algoritmo di merge richiede un numero di confronti ***proporzionale alla lunghezza degli array, O(n)***

Merge: esempio



- Si scandiscono i due vettori di ingresso, confrontandone le componenti a coppie
- Se in1[i] <= in2[j], out[k] = in1[i] (scrivi nella componente corrente del vettore out in1[i]); altrimenti, out[k]=in2[j]

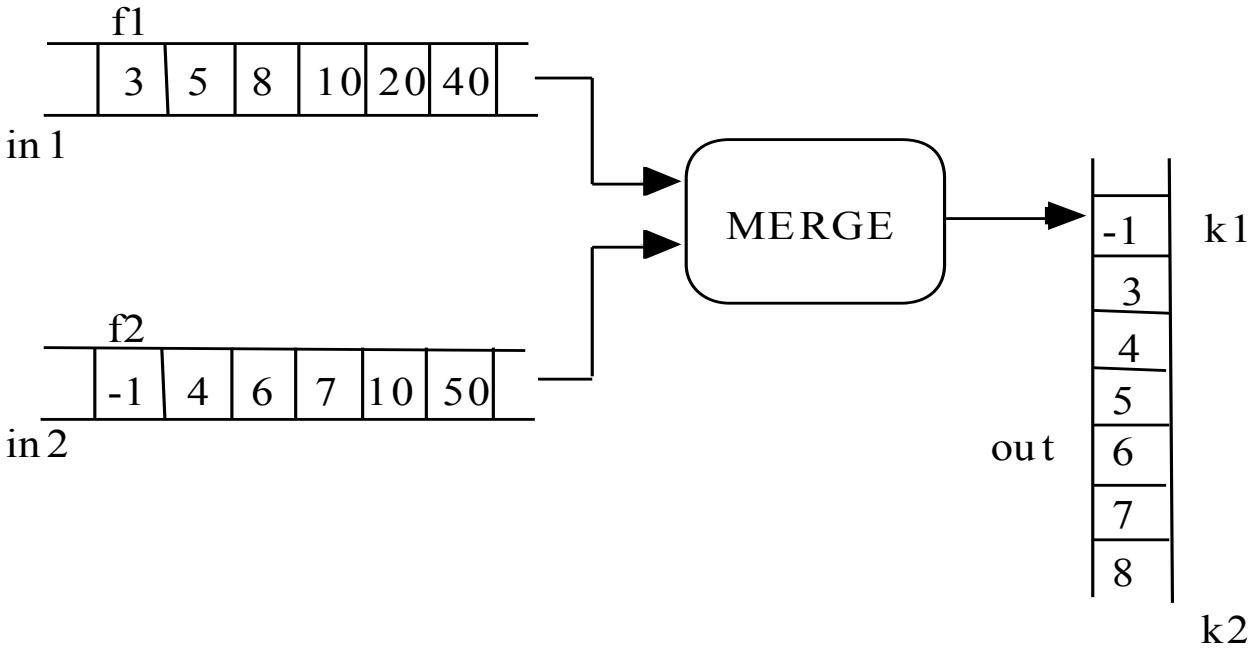
Algoritmo di Merge

- Indici **`i`**, **`j`** per scandire **`in1`** e **`in2`**, indice **`k`** per scrivere su **`out`**

- Si confrontano **`in1[i]`** e **`in2[j]`**:
  - se **`in1[i] <= in2[j]`**, scrive **`in1[i]`** nella componente k-esima di **`out`** (incrementa **`i`**, **`k`**)
  - altrimenti, scrive **`in2[j]`** nella componente k-esima di **`out`** (incrementa **`j`**, **`k`**)

- Se la scansione di uno dei vettori è arrivata all'ultima componente, si copiano i rimanenti elementi dell'altro nel vettore **`out`**

Merge: Complessità

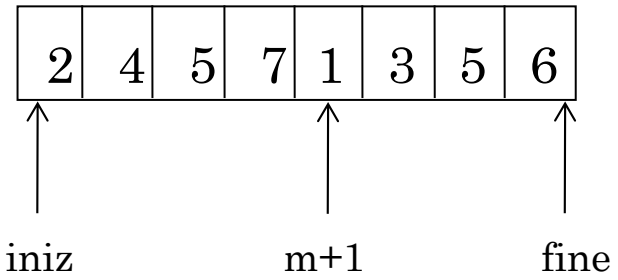


- Se **`in1`** e **`in2`** hanno ciascuno n componenti, nel caso peggiore, si eseguono:
  - 2n confronti
  - 2n copie
- L'algoritmo di fusione (merge) ha complessità asintorica lineare O(n)

Merge Sort

```
void merge(ARRAY a, int iniz1, int iniz2, int fine);
void mergeSortR(ARRAY a, int iniz, int fine);

void mergeSort(ARRAY a, int dim)
{
    mergeSortR(a, 0, dim - 1);
}

void mergeSortR(ARRAY a, int iniz, int fine)
{
    if (iniz < fine)
    {
        int m = (fine + iniz) / 2;
        mergeSortR(a, iniz , m);
        mergeSortR(a, m + 1, fine);
        merge(a, iniz, m + 1, fine);
    }
}
```

…mantiene uniformi le signature delle funzioni di ordinamento…



Merge

```
void merge(ARRAY a, int iniz1, int iniz2, int fine)
{
    static ARRAY aOut; /*vett. temporaneo*/
    int i, j, k;
    i = iniz1; j = iniz2; k = iniz1;
    while (i <= iniz2 - 1 && j <= fine) /*confronto: */
    {
        if (a[i] < a[j])
        {
            aOut[k] = a[i];
            i++;
        }
        else
        {
            aOut[k] = a[j];
            j++;
        }
        k++;
    }
```

continua…

Merge

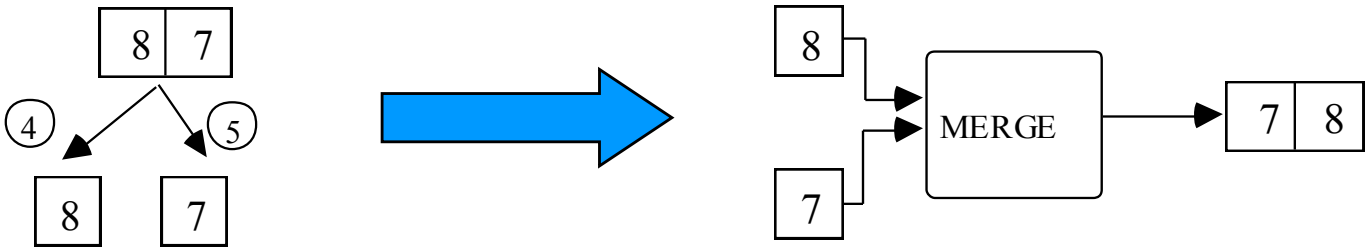
```
/* fasi di trattamento del vettore non terminato */
    while (i <= iniz2 - 1)
    {
        aOut[k] = a[i];
        i++;
        k++;
    }
    while (j <= fine)
    {
        aOut[k] = a[j];
        j++;
        k++;
    }
    /* copia da aOut in uscita */
    for (i = iniz1; i<= fine; i++)
        a[i] = aOut[i];
}
```

Merge Sort

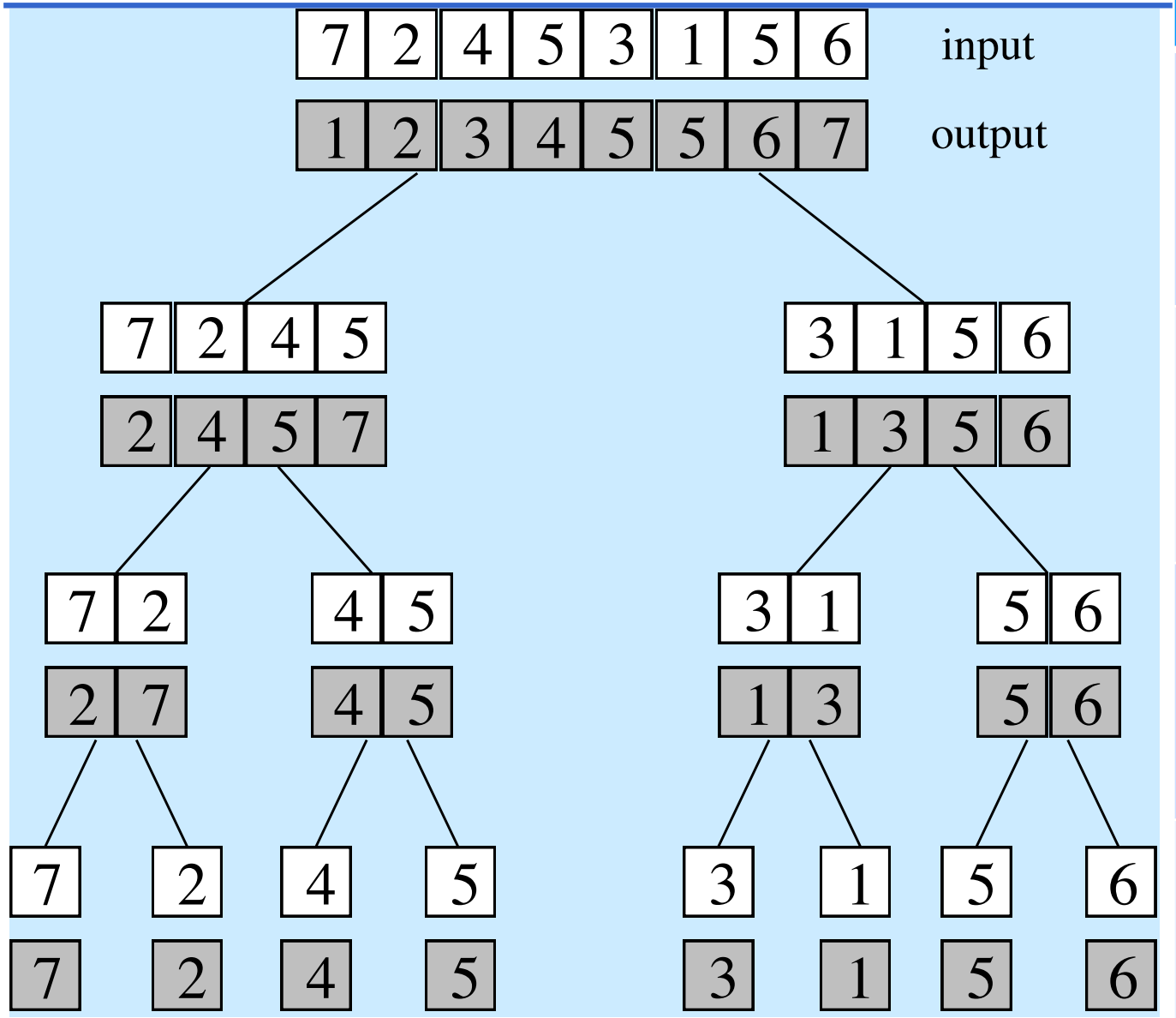
- È un algoritmo per sua natura ***ricorsivo***
- Il vettore di ingresso viene diviso in due sotto-vettori sui quali si richiama il merge sort
- Quando ciascun sotto-vettore è ordinato, i due vengono "fusi" attraverso la procedura di merge
- ***Si può dimostrare che Merge Sort è un algoritmo di ordinamento ottimo***
- Il numero di confronti effettuato da ***Merge Sort*** è una funzione che si comporta come $\mathbf{n*log_2 n}$ *vediamone il calcolo*

Merge Sort - Complessità

La condizione di terminazione della ricorsione è l'avere a che fare con array di lunghezza unitaria



Merge sort: complessità



livello	M	L
0	1	8 *(n)*
1	2	4 *(n/2)*
2	4	2 *(n/4)*
3 *(k)*	8 *(n)*	1 *($n/2^k$)*

Merge Sort - Costo

- Sia $n = 2^k$ (quindi $k = \log_2 n$)
- Ad ogni livello, si fondono M vettori di lunghezza L

livello	M	L
0	1	n
1	2	n/2
2	4	n/4
	…	
k-1	n/2	2
k	n	1

- Le attivazioni di *merge* eseguono L*M (sempre uguale a n) confronti (la complessità asintotica del merge è lineare nelle dimensioni dei vettori che fonde), per ciascun livello (i livelli sono k)
- Il numero totale di confronti è quindi: n*k = **$O(n \log_2 n)$**

Ordinamenti ottimi

- Merge sort
- Quick sort
- Presentare la funzione **`qsort`** della libreria **`stdlib.h`** e il suo utilizzo

Quick Sort

- Come merge-sort, suddivide il vettore in due sotto-array, delimitati da un elemento "sentinella" (***pivot***)
- Il pivot viene spostato in modo opportuno in modo da raggiungere…
- …l'**obiettivo** che è quello di avere nel primo sotto-array solo elementi minori o uguali al pivot, nel secondo sotto-array solo elementi maggiori

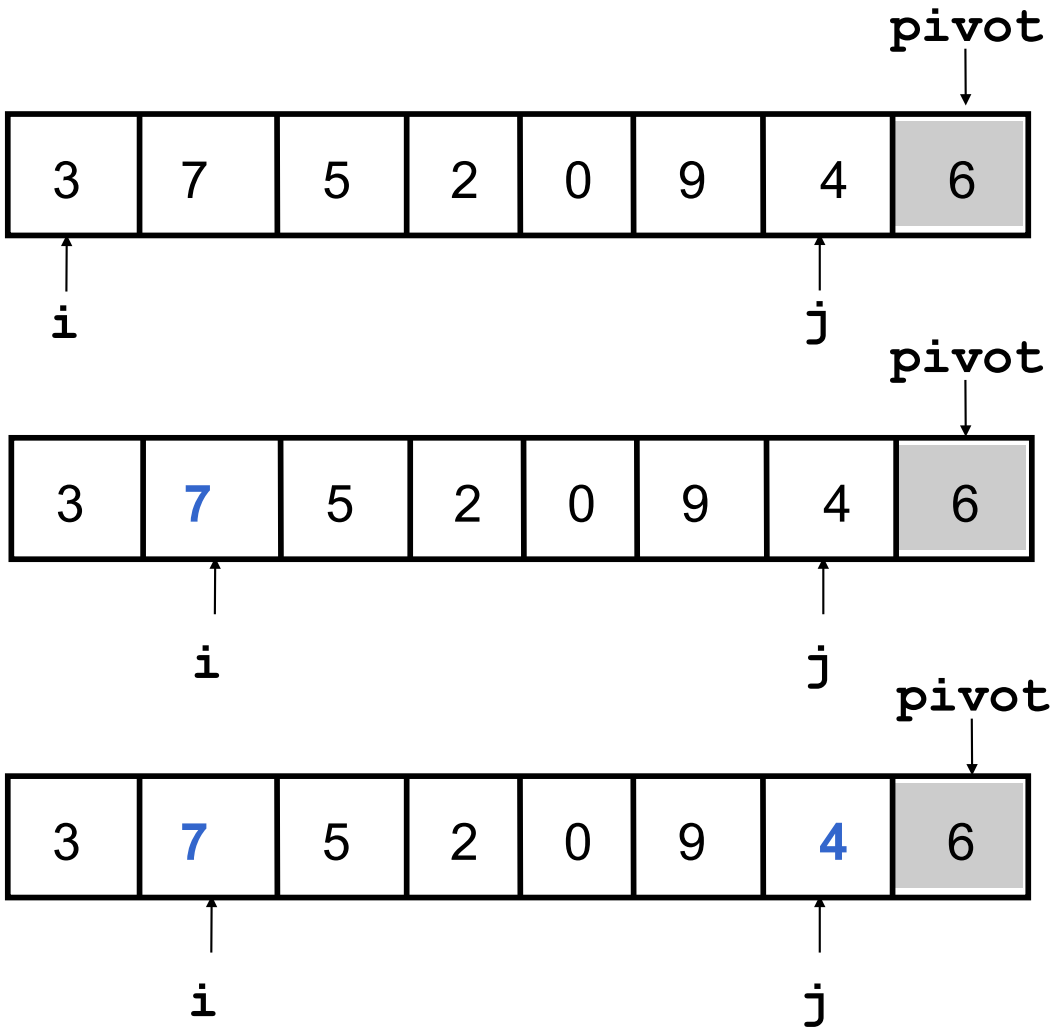
Quick Sort - Algoritmo

- Si determina arbitrariamente un pivot
  - ad esempio **`pivot = a[dim - 1]`**
- Si scandisce il vettore dato mediante due indici:
  - **`i`**, che parte da **`0`** e procede in **avanti**
  - **`j`**, che parte da **`dim - 1`** (**`dim`** = dimensione del vettore) e procede all'**indietro**
- **Scansione in avanti:**
  - ogni elemento **`a[i]`** viene confrontato con il **`pivot`**
    se **`a[i] > pivot`**, la scansione in avanti si ferma e si passa alla…
- **Scansione all'indietro:**
  - ogni elemento **`a[j]`** viene confrontato con il **`pivot`**
    se **`a[j] < pivot`**, la scansione in indietro si ferma e l'elemento **`a[j]`** viene scambiato con **`a[i]`**

Quick Sort – Algoritmo

- Poi si riprende con la scansione avanti, indietro, ... Il tutto si ferma quando **`i == j`**. A questo punto si scambia **`a[i]`** con il **`pivot`**
- Alla fine della scansione il **`pivot`** è collocato nella sua posizione definitiva
- L'algoritmo è ***ricorsivo***: si richiama su ciascun sotto-array fino a quando non si ottengono sotto-array con un solo elemento
- A questo punto il vettore iniziale risulta ordinato

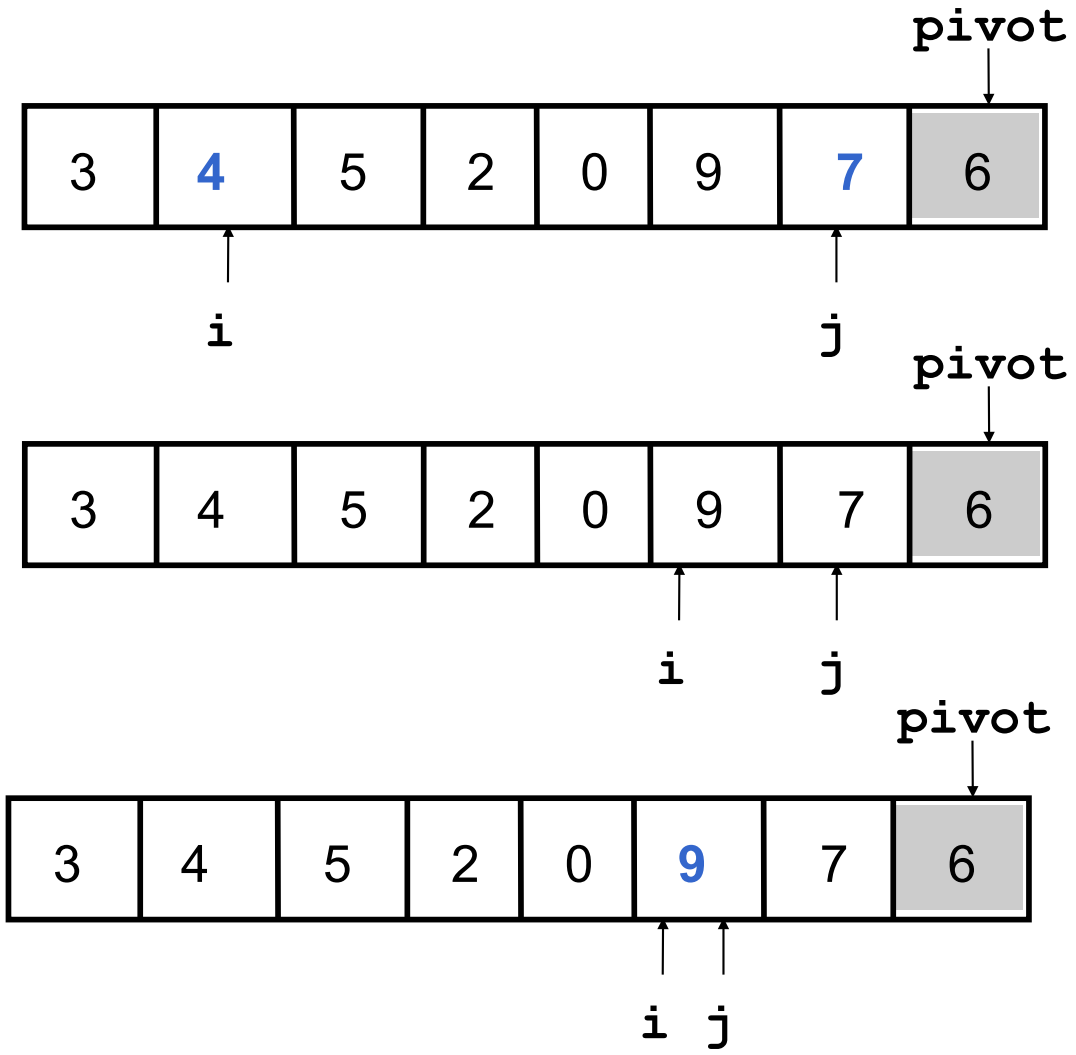
Quick Sort



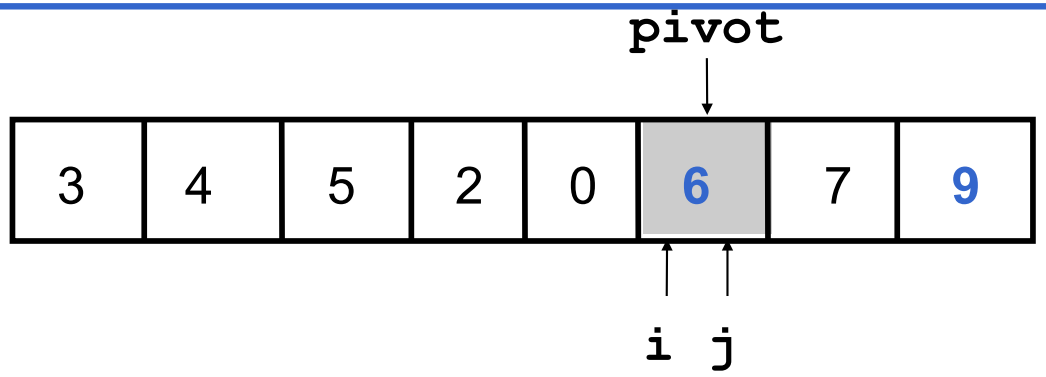
Scansione in avanti

Scansione all'indietro

Quick Sort


pivot
3
4
5
2
0
9
7
6
i
j
Scambio
pivot
3
4
5
2
0
9
7
6
i
j
Scansione in avanti
pivot
3
4
5
2
0
9
7
6
i
j
Scansione all' indietro

Quick Sort



Fine scansioni – scambio con il pivot

- Il pivot è nella posizione definitiva

- Ripetere il procedimento sui due sotto-array
  - `a[0, i - 1]`
  - `a[i + 1, dim - 1]`

Quick Sort

- Ancora una volta, per uniformare le signature si introduce una funzione che fa da interfaccia con i clienti e che invoca opportunamente la funzione ricorsiva

```
void quickSortR(ARRAY a, int iniz, int fine);

void quickSort(ARRAY a, int dim)
{
    quickSortR(a, 0, dim - 1);
}
```

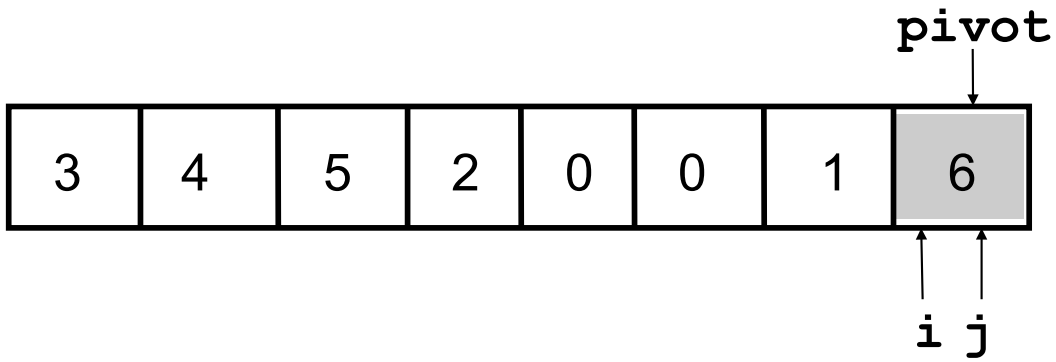
Quick Sort

```
void quickSortR(ARRAY a, int iniz, int fine)
{
    int i, j, iPivot;
    ELEMENT pivot;
    if (iniz < fine)
    {
        i = iniz;
        j = fine;
        iPivot = fine;
        pivot = a[iPivot];
        do  /* trova la posizione del pivot */
        {
            while (i < j && a[i] <= pivot) i++;
            while (j > i && a[j] >= pivot) j--;
            if (i < j) swap(&a[i], &a[j]);
        }
        while (i < j);
```

continua…

Quick Sort

```
        /* determinati i due sottoinsiemi */
        /* posiziona il pivot */

        if (i != iPivot && a[i] != a[iPivot])
        {
            swap(&a[i], &a[iPivot]);
            iPivot = i;
        }

        /* ricorsione sulle sottoparti, se necessario */
        if (iniz < iPivot - 1)
            quickSortR(a, iniz, iPivot - 1);
        if (iPivot + 1 < fine)
            quickSortR(a, iPivot + 1, fine);

    } /* (iniz < fine) */
} /* quickSortR */
```


pivot
3
4
5
2
0
0
1
6
i
j

Quick sort- Costo

- Ogni attivazione di **`quickSortR`** effettua un numero di confronti proporzionale alla dimensione del vettore su cui opera
- Il numero di attivazioni di **`quickSortR`** dipende dalla scelta del pivot.
- ***Caso peggiore***: si sceglie un pivot tale che un sottovettore ha lunghezza nulla (n-1 passi)
- Numero di confronti al passo i, pari a n-i (dimensione dell'altro sottovettore)

$$O(n^2)$$

Quick sort- Costo

- ***Caso migliore***: si sceglie un pivot tale che il vettore si dimezza ($\log_2$ n passi)
- Numero di confronti al passo i, pari a n (2^i vettori di lunghezza $n/2^i$)

$O(n*\log_2 n)$

- ***Caso medio***: la scelta casuale del pivot rende probabile la divisione in due sotto-vettori aventi circa lo stesso numero di elementi
- Stessa complessità del caso migliore

Quick Sort

- Il Quick Sort è efficiente come il Merge Sort se il pivot è sempre il valore baricentrico tra quelli da ordinare
- Se si ha sfortuna allora l'efficienza scende fino ad un livello compatibile con il Bubble Sort (che non è proprio un fulmine…)
- Come considerazione generale, l'implementazione vista si comporta bene per array molto disordinati
- Scegliere l'ultimo elemento come pivot può essere una scelta rischiosa: se l'ultima parte dell'array è già ordinata?
- Sarebbe meglio scegliere il pivot a "caso" fra gli elementi a disposizione → però la scelta a caso può costare (in termini di prestazioni) di più che scegliere male il pivot…

Quick Sort

- E' un algoritmo ottimo (caso migliore e caso medio)
- E' disponibile in C come funzione di libreria
- Parametrica rispetto alla operazione/funzione di confronto (che implementa la relazione d'ordine) in base alla quale ordinare una batteria di celle

qsort

- Funzione generica di libreria (**stdlib.h**):

```
void qsort(void *base, size_t n, size_t width,
           int (*fcmp)(void *el1,void *el2) );
```

- **base** punta al primo elemento del vettore da ordinare
- **size_t** è un tipo definito in **stdlib.h** (in genere un **unsigned long**)
- **n** è il numero di elementi nel vettore
- **width** è la dimensione di ogni elemento del vettore in byte
- **fcmp** è la funzione di confronto con
  - **Argomenti** puntatori ai due elementi del vettore da confrontare, e che dà come **risultato**:
  - Se *e1 < *el2 un intero < 0
  - Se *el1 == *el2 0
  - Se *el1 > *el2 un intero > 0

Quick Sort – esempio di chiamata

```
#define N   11
typedef float ELEMENT;
typedef ELEMENT ARRAY[N];
typedef enum {false, true} Boolean;

void quickSortR(ARRAY a, int iniz, int fine);

void quickSort(ARRAY a, int dim)
{   quickSortR(a, 0, dim - 1); }

. . .
ARRAY V;  /* init V */
. . .
quickSortR(V, 0, N-1);
/* quickSortR e' specifico per i float */
```

Quick Sort – funzione generica

```
#define N  11
typedef float ELEMENT;
typedef ELEMENT ARRAY[N];
typedef enum {false, true} Boolean;

int fcompare (ELEMENT *el1, ELEMENT *el2)
{   if(*el1==*el2) return 0;
     else if(*el1<*el2) return -1;
           else return 1;}
. . .
ARRAY V;  /* init V */
qsort(V, N, sizeof(ELEMENT), fcompare);
```

Esercizio (laboratorio): `qsort`

- Ordiniamo i **`MAXDIM`** elementi di un vettore di reali V, letti da input

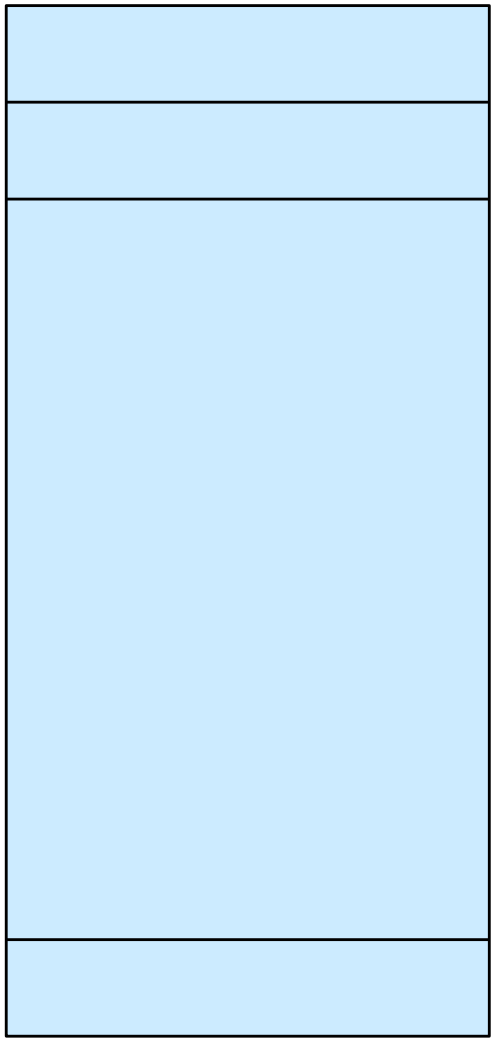
```
#define MAXDIM  11
typedef float ELEMENT;
typedef ELEMENT ARRAY[MAXDIM];
```

- base ⇒ V (che coincide con &V[0])
- N ⇒ **`MAXDIM`**
- width ⇒ sizeof(V[0]) oppure sizeof(float)
- fcmp deve essere definita

```
int compare(ELEMENT *a, ELEMENT *b);
```

qsort

- base ⇒ &V[0]
- N ⇒ **MAXDIM**
- width ⇒ sizeof(V[0])
- fcmp deve essere definita



ESERCIZIO (in Laboratorio)

- È dato un vettore di dimensione N+k contenente N numeri interi (N può essere anche 0), ordinati in senso non decrescente

- Ricevendo uno alla volta k interi, li si inserisca nel vettore, mantenendo l'ordinamento del vettore ad ogni passo di inserimento

- Determinare le condizioni che generano il maggior numero di confronti tra elementi, in funzione di N e k