Ricerca di un elemento in un Array

Sapendo che il vettore è ***ordinato***, la ricerca può essere ottimizzata

- ***Vettore ordinato in senso non decrescente:***

  Esiste una relazione d'ordine totale sul dominio degli elementi

2	3	5	5	7	8	10	11

  `se i<j` si ha **`V[i]<=V[j]`**

- ***Vettore ordinato in senso crescente:***

2	3	5	6	7	8	10	11

  `se i<j` si ha **`V[i]<V[j]`**

In modo analogo si definiscono l'ordinamento in senso ***non crescente*** e ***decrescente***

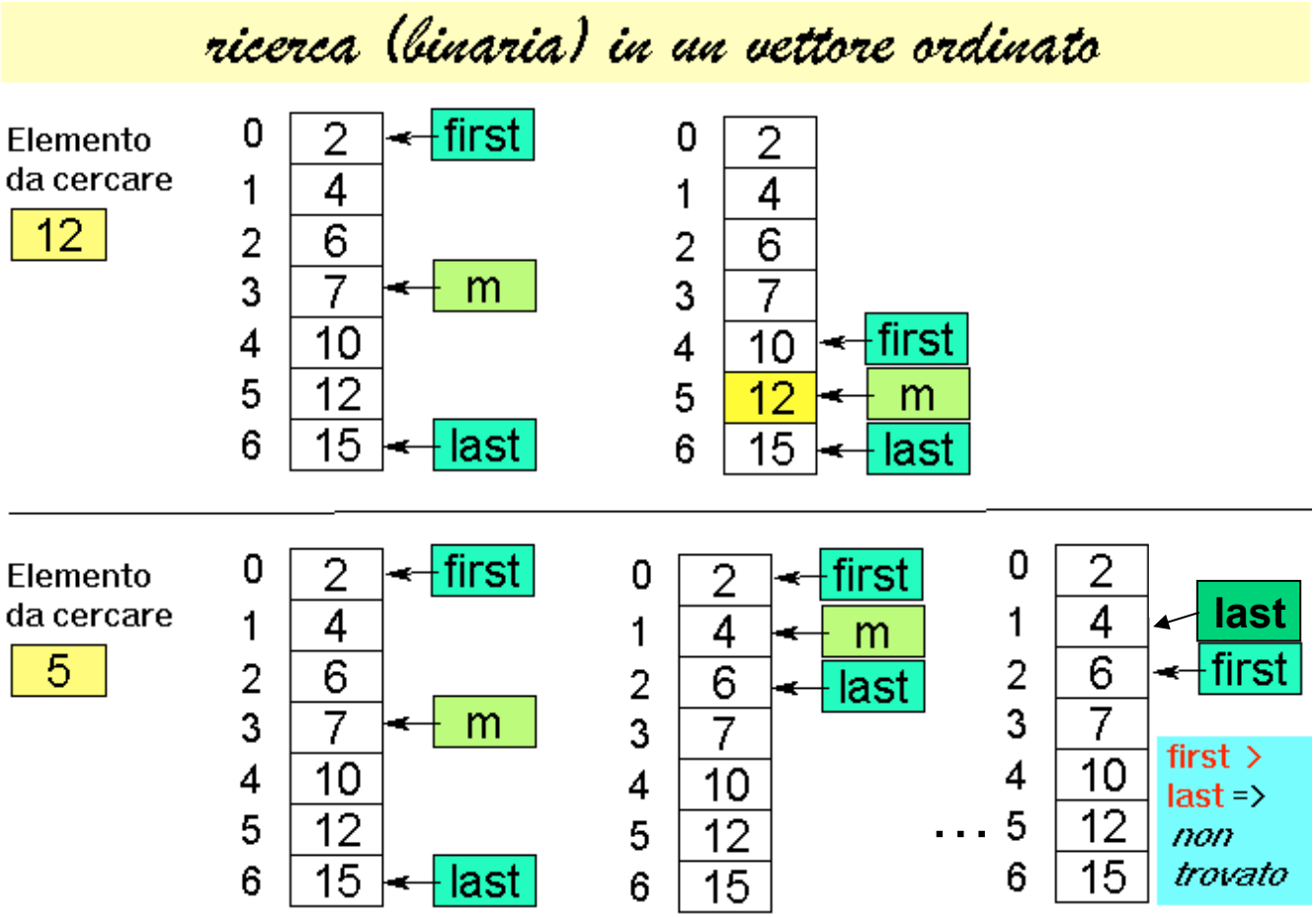
RICERCA BINARIA

Ricerca binaria di un elemento in un vettore ordinato in senso non decrescente in cui il primo elemento è `first` e l'ultimo `last`

La tecnica di ***ricerca binaria***, rispetto alla ricerca esaustiva, consente di ***eliminare ad ogni passo metà degli elementi del vettore***

RICERCA BINARIA

Esempio



RICERCA BINARIA

- Si confronta l'elemento cercato **`el`** con quello mediano del vettore, **`V[med]`**
- Se **`el==V[med]`**, fine della ricerca (**`trovato=true`**)
- Altrimenti, se il vettore ha almeno due componenti (**`first<=last`**):
  - se **`el<V[med]`**, ripeti la ricerca nella prima metà del vettore (indici da **`first`** a **`med-1`**)
  - se **`el>V[med]`**, ripeti la ricerca nella seconda metà del vettore (indici da **`med+1`** a **`last`**)

RICERCA BINARIA

```
int ricerca_bin (int vet[], int el)
{int first=0, last=N-1, med=(first+last)/2;
 int T=0;
 while ((first<=last)&&(T==0))
  { if (el==vet[med])
          T=1;
    else
       if (el < vet[med])  last=med-1;
       else   first=med+1;
    med = (first + last) / 2;
  }
 return  T;
}
```

Ricerca binaria di un elemento

```
#include <stdio.h>
#define N 15

int ricerca_bin (int vet[], int el);
main ()
{int i;
 int a[N];
 printf ("Scrivi %d numeri interi ordinati\n", N);
 for (i = 0; i<N; i++)
     scanf ("%d", &a[i]);
 printf ("Valore da cercare: ");
 scanf ("%d",&i);
 if (ricerca_bin(a,i)) printf("\nTrovato\n");
     else printf("\nNon trovato\n");
}
```

OSSERVAZIONI

Si noti che la ricerca binaria può essere definita facilmente in ***modo ricorsivo***

Si noti infatti che si effettua un ***confronto dell'elemento cercato `el` con l'elemento di posizione media del vettore `V[med]`***

- Se l'elemento cercato è uguale si termina (caso base)
- Altrimenti se **`el < V[med]`** si effettua una ricerca binaria sulla prima metà del vettore
- Altrimenti (**`se el > V[med]`**) si effettua una ricerca binaria sulla seconda metà del vettore

Esercizio: si scriva procedura per ***ricerca binaria ricorsiva***

ALGORITMI DI ORDINAMENTO

- **Scopo:** ***ordinare una sequenza di elementi*** **in base a una certa** ***relazione d'ordine***
  - **lo scopo finale è ben definito**
    → ***algoritmi equivalenti***
  - **diversi algoritmi possono avere** ***efficienza assai diversa***
- **Ipotesi:**
  gli elementi siano memorizzati in un array.

ALGORITMI DI ORDINAMENTO

Principali algoritmi di ordinamento:

- ***naïve sort*** (semplice, intuitivo, poco efficiente)
- ***bubble sort*** (semplice, un po' più efficiente)
- ***insert sort*** (intuitivo, abbastanza efficiente)
- ***merge sort*** (non intuitivo, molto efficiente)
- ***quick sort*** (non intuitivo, alquanto efficiente)

Per "misurare le prestazioni" di un algoritmo, conteremo quante volte viene svolto il ***confronto fra elementi dell'array.***

NAÏVE SORT

- **Molto intuitivo e semplice, è il primo che viene in mente**

  Specifica (sia **`n`** la dimensione dell'array **`v`**)

  `while` **`(`***<array non vuoto>***`)`** **`{`**

  <trova la posizione ***`p`*** *del massimo>*

  `if (p<n-1)` *<scambia* ***`v[n-1]`*** *e* ***`v[p]`*** *>*

  /* ora v[n-1] contiene il massimo */

  <restringi l'attenzione alle prime ***`n-1`*** *caselle dell' array, ponendo* ***`n' = n-1`*** *>*

  `}`

NAÏVE SORT

Codifica

```
void naiveSort(int v[], int n){
 int p;
 while (n>1) {
   p = trovaPosMax(v,n);
   if (p<n-1) scambia(&v[p],&v[n-1]);
   n--;
 }
}
```

La dimensione dell'array cala di 1 a ogni iterazione

NAÏVE SORT

Codifica

```
int trovaPosMax(int v[], int n){
   int i, posMax=0;
   for (i=1; i<n; i++)
      if (v[posMax]<v[i]) posMax=i;
   return posMax;
}
```

All'inizio si assume v[0] come max di tentativo.

Si scandisce l'array e, se si trova un elemento maggiore del max attuale, lo si assume come nuovo max, memorizzandone la posizione.

NAÏVE SORT

Valutazione di complessità

- Il numero di *confronti* necessari vale *sempre*:

$$\mathbf{(N-1) + (N-2) + (N-3) + \ldots + 2 + 1 =}$$
$$\mathbf{= N*(N-1)/2 \;\; = \;\; O(N^2/2)}$$

- *Nel caso peggiore*, questo è anche il numero di scambi necessari (in generale saranno meno)
- ***Importante*: la complessità non dipende dai particolari dati di ingresso**
  - **l'algoritmo fa gli stessi confronti sia per un array disordinato, sia per un array *già ordinato!!***

BUBBLE SORT (ordinamento a bolle)

- **Corregge il difetto principale del naïve sort: quello di *non accorgersi se l'array, a un certo punto, è già ordinato.***
- **Opera per *"passate successive"* sull'array:**
  - **a ogni iterazione, considera una ad una *tutte le possibili coppie di elementi adiacenti*, scambiandoli se risultano nell'ordine errato**
  - **così, dopo ogni iterazione, l'elemento massimo è in fondo alla parte di array considerata**
- **Quando non si verificano scambi, l'array è ordinato, e l'algoritmo termina.**

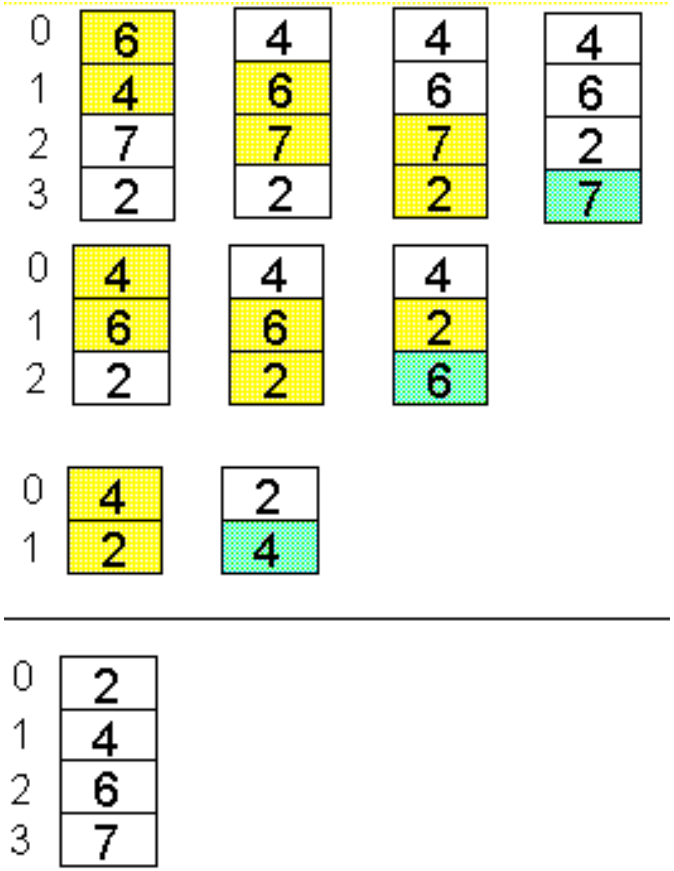
BUBBLE SORT

Codifica

```
void bubbleSort(int v[], int n){
  int i; int ordinato = 0;
  while (n>1 && ordinato==0){
    ordinato = 1;
    for (i=0; i<n-1; i++)
      if (v[i]>v[i+1]) {
        scambia(&v[i],&v[i+1]);
        ordinato = 0; }
    n--;
  }
}
```

BUBBLE SORT

Esempio



Iª passata (dim. = 4)
al termine, 7 è a posto.

IIª passata (dim. = 3)
al termine, 6 è a posto.

IIIª passata (dim. = 2)
al termine, 4 è a posto.

array ordinato

BUBBLE SORT

Valutazione di complessità

- Caso peggiore: numero di *confronti* identico al precedente → **$O(N^2/2)$**
- ***Nel caso migliore, però, basta una sola passata***, con N-1 confronti → **O(N)**
- *Nel caso medio,* i confronti saranno compresi fra N-1 e $N^2/2$, a seconda dei dati di ingresso.

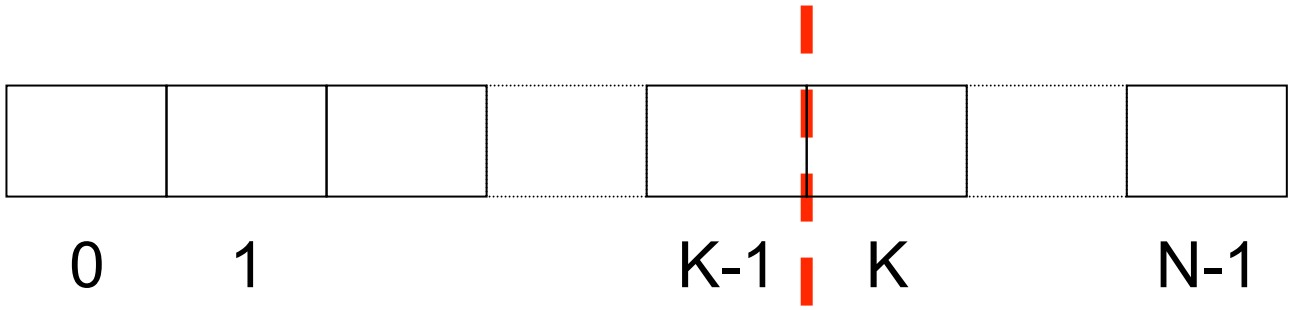
INSERT SORT

- **Per ottenere un array ordinato basta *costruirlo ordinato,* *inserendo gli elementi al posto giusto* *fin dall'inizio.***
- **Idealmente, il metodo costruisce un nuovo array, contenente gli stessi elementi del primo, ma ordinato.**
- **In pratica, *non è necessario costruire un secondo array*, in quanto le stesse operazioni possono essere svolte direttamente sull'array originale: così, alla fine esso risulterà ordinato.**

INSERT SORT

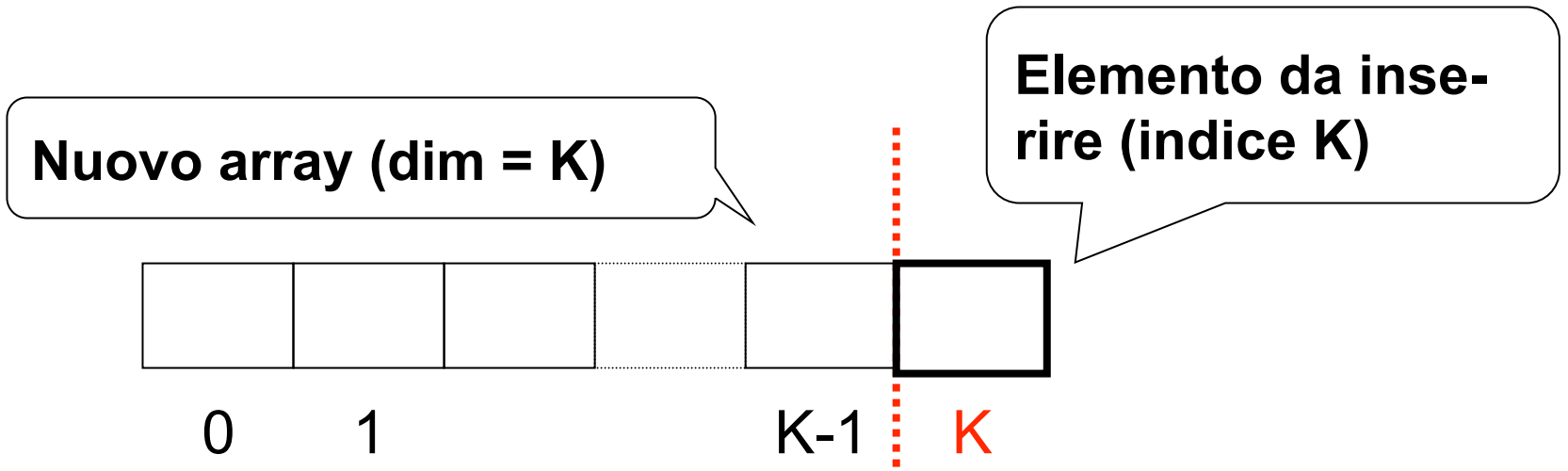
Scelta di progetto

- ***"vecchio" e "nuovo" array condividono lo stesso array fisico di N celle (da 0 a N-1)***
- ***in ogni istante, le prime K celle (numerate da 0 a K-1) costituiscono il nuovo array***
- ***le successive N-K celle costituiscono la parte residua dell'array originale***



INSERT SORT

- Come conseguenza della scelta di progetto fatta, **in ogni istante *il nuovo elemento da inserire si trova nella cella successiva alla fine del nuovo array, cioè* la (K+1)-esima** (il cui indice è K)



INSERT SORT

Specifica

```
for (k=1; k<n; k++)
```

<inserisci alla posizione k-esima del nuovo array ***l'elemento minore fra quelli rimasti nell'array originale***>

All'inizio (k=1) il nuovo array è la sola prima cella

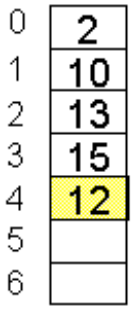
Codifica

```
void insertSort(int v[], int n){
   int i;
   for (k=1; k<n; i++)
      insMinore(v,k);
}
```

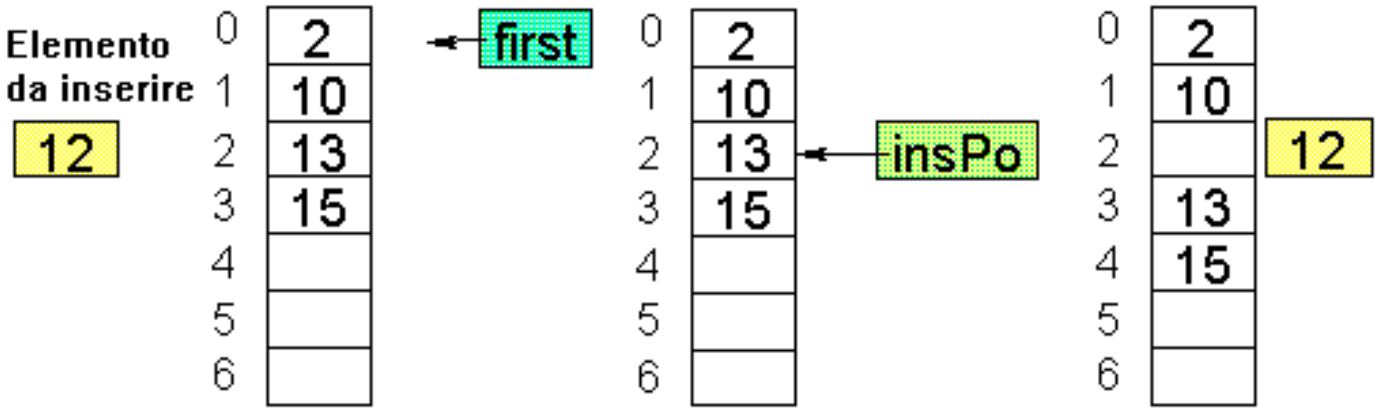
Al passo *k*, la demarcazione fra i due array è alla posizione k

INSERT SORT

Esempio



Scelta di progetto: se il nuovo array è lungo K=4 (numerate da 0 a 3) l'elemento da inserire si trova nella cella successiva (di indice K=4).



INSERT SORT

Specifica di insMinore()

```
void insMinore(int v[], int pos){
```

<determina la posizione in cui va inserito il nuovo elemento>

<crea lo spazio spostando gli altri elementi in avanti di una posizione>

<inserisci il nuovo elemento alla posizione prevista>

```
}
```

INSERT SORT

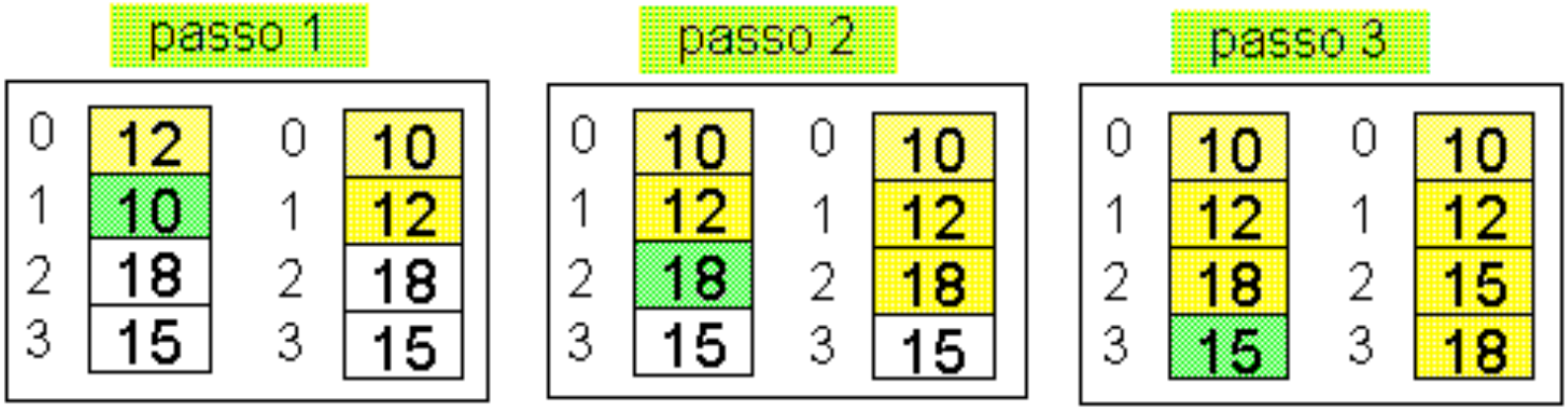
Codifica di insMinore()

```
void insMinore(int v[], int pos){
 int i = pos-1, x = v[pos];
  while (i>=0 && x<v[i])
{
   v[i+1]= v[i];          /* crea lo
 spazio */
   i--;
  }
  v[i+1]=x;  /* inserisce l'elemento */
}
```

Determina la posizione a cui inserire x

INSERT SORT

Esempio



INSERT SORT

Valutazione di complessità

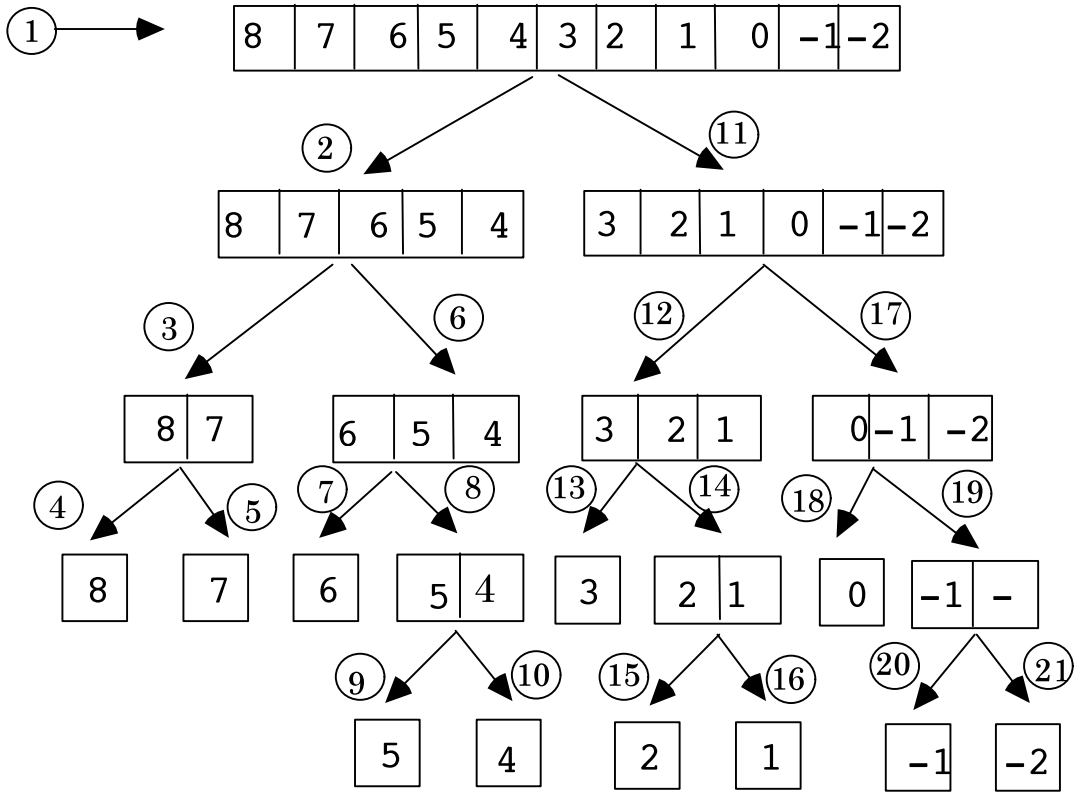
- *Nel caso peggiore* (array ordinato al contrario), richiede 1+2+3+…+(N-1) confronti e spostamenti → **$O(N^2/2)$**
- *Nel caso migliore* (array già ordinato), bastano solo N-1 confronti (senza spostamenti)
- ***Nel caso medio***, a ogni ciclo il nuovo elemento viene inserito nella posizione centrale dell'array → 1/2+2/2+…+(N-1)/2 confronti e spostamenti
  Morale: $O(N^2/4)$

MERGE SORT

- **Algoritmo del tipo “divide et impera” per sua natura ricorsivo**
- ***In pratica:***
  - **si spezza l’array in due parti *di ugual dimensione***
  - **si ordinano separatamente queste due parti *(chiamata ricorsiva)***
  - **si fondono i due sub-array ordinati così ottenuti in modo da ottenere un unico array ordinato.**
- **Il punto cruciale è l’algoritmo di fusione (*merge*) dei due array**

MERGE SORT

Esempio



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

MERGE SORT

Specifica

```
void mergeSort(int v[], int iniz, int fine,
                        int vout[]) {
  if (<array non vuoto>){
   <partiziona l'array in due metà>
   <richiama mergeSort ricorsivamente sui due sub-array,
    se non sono vuoti>
   <fondi in vout i due sub-array ordinati>
  }
}
```

MERGE SORT

Codifica

```
void mergeSort(int v[], int iniz, int fine,
                        int vout[]) {
 int mid;
 if ( iniz < fine ) {
   mid = (fine + iniz) / 2;
    mergeSort(v, iniz, mid,  vout);
    mergeSort(v, mid+1, fine, vout);
    merge(v, iniz, mid+1, fine, vout);
  }
}
```

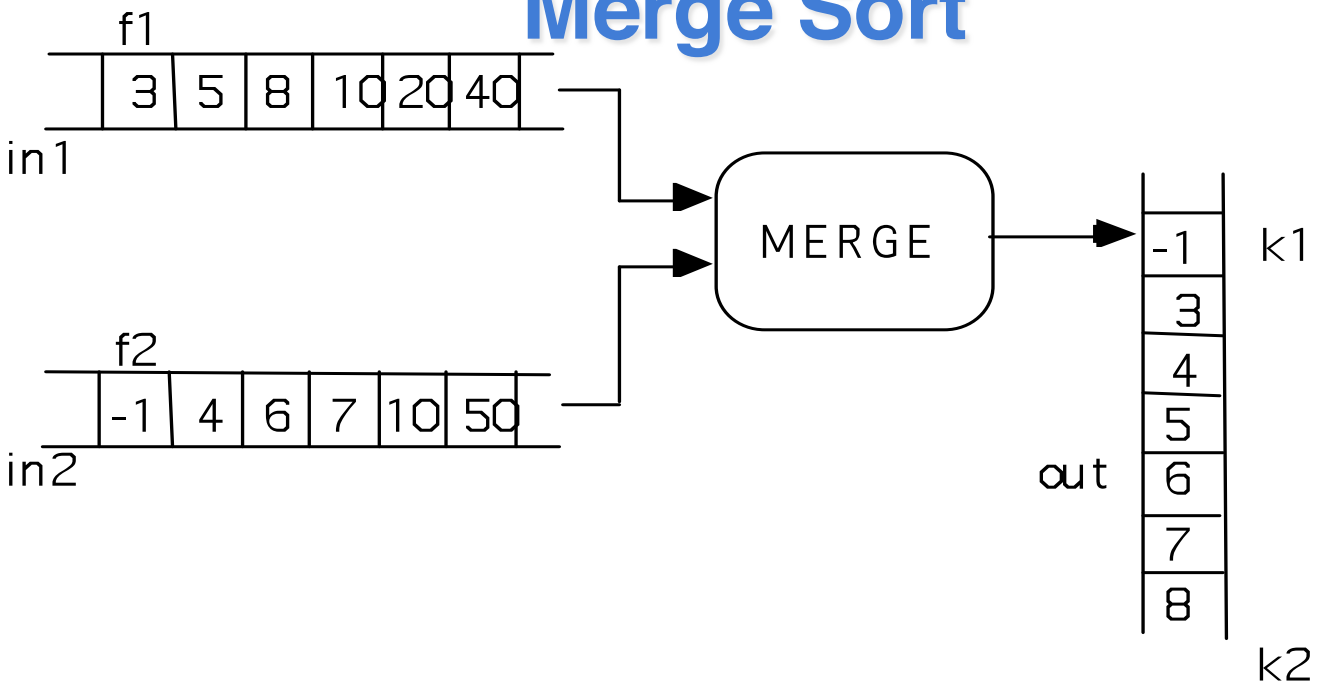
mergeSort() si limita a suddividere l'array: è merge() che svolge il lavoro

Merge Sort

- Ordinamento per **fusione**

- Utilizza un algoritmo di **Merge**
  - Dati due vettori x, y con m componenti ciascuno e ordinati in ordine crescente, produrre un unico vettore z, di 2*m componenti che sia ordinato
  - Algoritmo di merge richiede un numero di passi proporzionale alla lunghezza degli array

Merge Sort



- 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]

MERGE SORT

Codifica di merge()

```
void  merge(int v[], int i1, int i2,
  int fine, int vout[]){
  int i=i1, j=i2, k=i1;
  while ( i <= i2-1 &&  j <= fine ) {
     if (v[i] < v[j]) {vout[k] = v[i]; i++;}
     else {vout[k] = v[j]; j++;}
     k++;
  }
  while (i<=i2-1){vout[k]=v[i]; i++; k++;}
  while (j<=fine){vout[k]=v[j]; j++; k++;}
  for (i=i1; i<=fine; i++) v[i] = vout[i];
}
```

Merge Sort - Costo

- Il costo dell'operazione di Merge è proporzionale alla lunghezza degli array in ingresso → O(n)
- La funzione Merge Sort richiama se stessa per due volte sulla metà dell'array in ingresso → è possibile associare al tempo di esecuzione di Merge Sort la funzione temporale:

  $T(n) = 2T(n/2) + O(n)$

  Si può dimostrare che tale funzione è

  $O(n \log(n))$

QUICK SORT

- **Idea base:** ***ordinare un array corto è molto meno costoso che ordinarne uno lungo*****.**
- **Conseguenza: può essere utile** ***partizionare l'array in due parti, ordinarle separatamente*****, e infine** ***fonderle insieme.***
- **<u>In pratica</u>:**
  - **si suddivide il vettore in due "sub-array", delimitati da un elemento "sentinella" (*pivot*)**
  - **il primo array deve contenere solo elementi *minori o uguali* al pivot, il secondo solo elementi *maggiori* del pivot.**

QUICK SORT

Algoritmo ricorsivo:

- **i due sub-array ripropongono un problema di ordinamento *in un caso più semplice* (array più corti)**
- **a forza di scomporre un array in sub-array, si giunge a un array di un solo elemento, che è già ordinato (*caso banale*).**

QUICK SORT

- **Idea base:** ***ordinare un array corto è molto meno costoso che ordinarne uno lungo*****.**
- **Conseguenza: può essere utile *partizionare l'array in due parti, ordinarle separatamente*, e infine *fonderle insieme.***
- **In pratica:**
  - **si suddivide il vettore in due "sub-array", delimitati da un elemento "sentinella" (*pivot*)**
  - **il primo array deve contenere solo elementi *mi-nori o uguali* al pivot, il secondo solo elementi *maggiori* del pivot.**
- **Alla fine di questo blocco di lucidi c'e' il codice ma non lo vedremo a lezione e non fa parte del programma.**

QUICK SORT

- **Si può dimostrare che $O(N \log_2 N)$ è un limite inferiore alla complessità del *problema dell'ordinamento di un array.***
- **Dunque, *nessun algoritmo, presente o futuro, potrà far meglio di* $O(N \log_2 N)$**
- **Però, il quicksort raggiunge questo risultato *solo se il pivot è scelto bene***
  - **per fortuna, la suddivisione in sub-array uguali è la cosa più probabile nel caso medio**
  - **l'ideale sarebbe però che tale risultato fosse raggiunto *sempre:* a ciò provvede il *Merge Sort.***

QUICK SORT

Struttura dell'algoritmo

- **scegliere un elemento come pivot**
- **partizionare l'array nei due sub-array**
- **ordinarli separatamente (*ricorsione*)**

L'operazione-base è il *partizionamento dell'array nei due sub-array*. Per farla:

- se il primo sub-array ha un elemento > pivot, e il secondo array un elemento < pivot, questi due elementi vengono *scambiati*

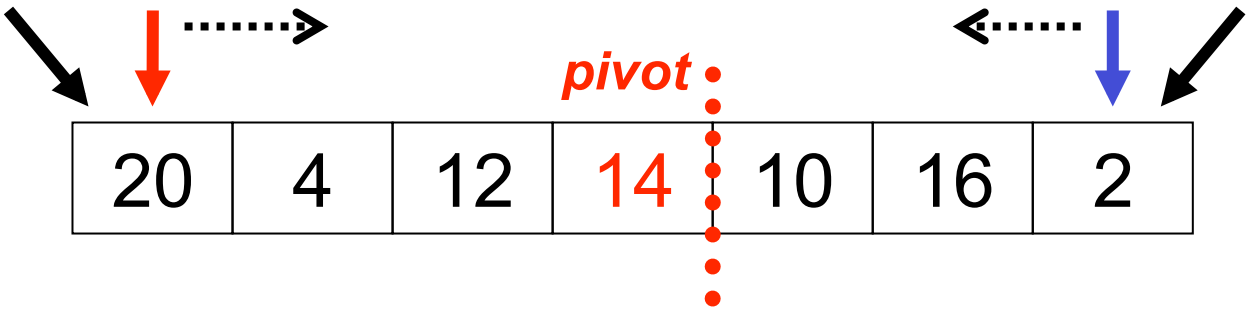
Poi si riapplica quicksort ai due sub-array.

QUICK SORT

Esempio: *legenda*

freccia rossa (i): indica l'inizio del II° sub-array

freccia blu (j): indica la fine del I° sub-array

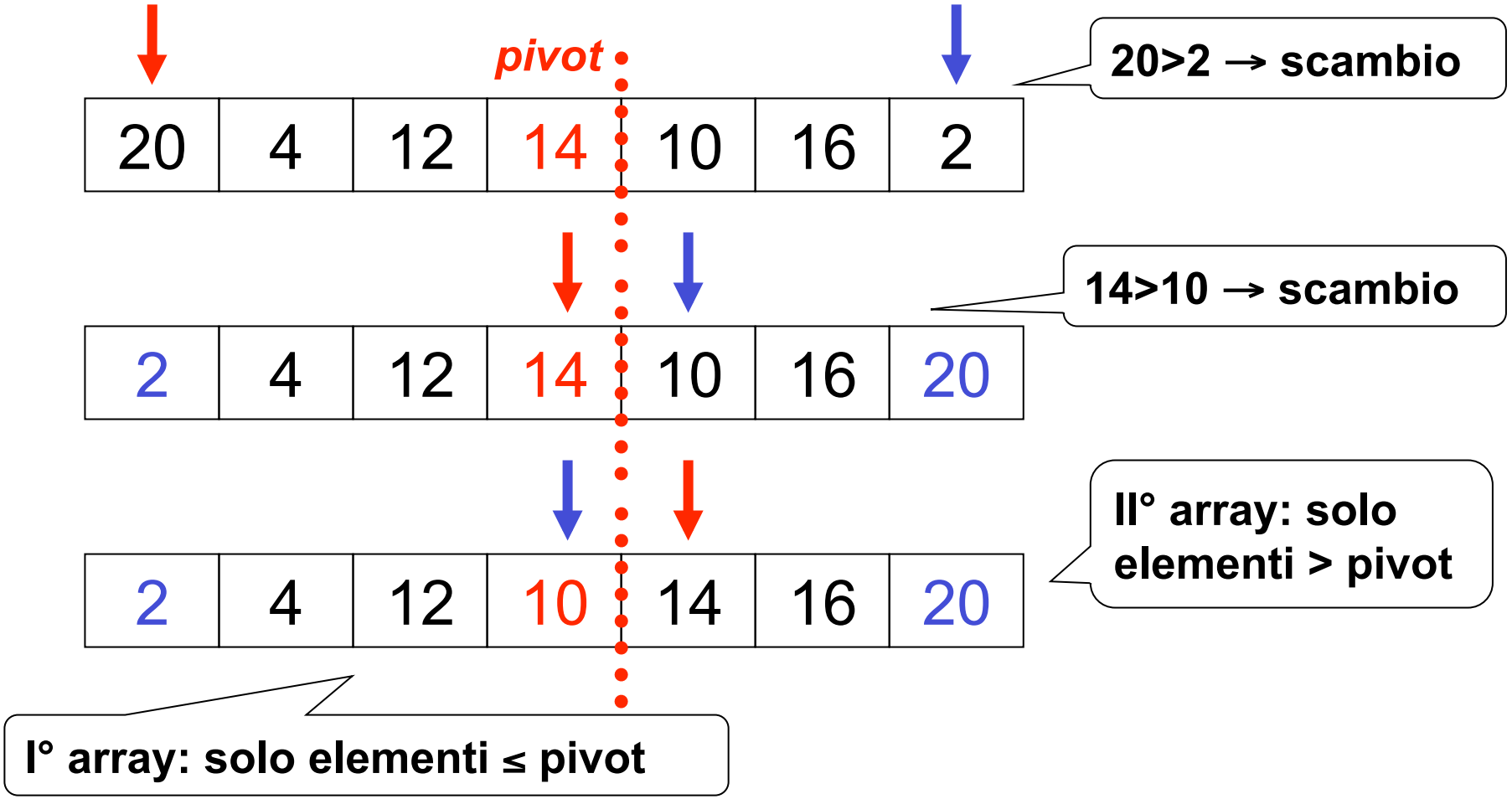


freccia nera (iniz): indica l'inizio dell'array (e del I° sub-array)

freccia nera (fine): indica la fine dell'array (e del II° sub-array)

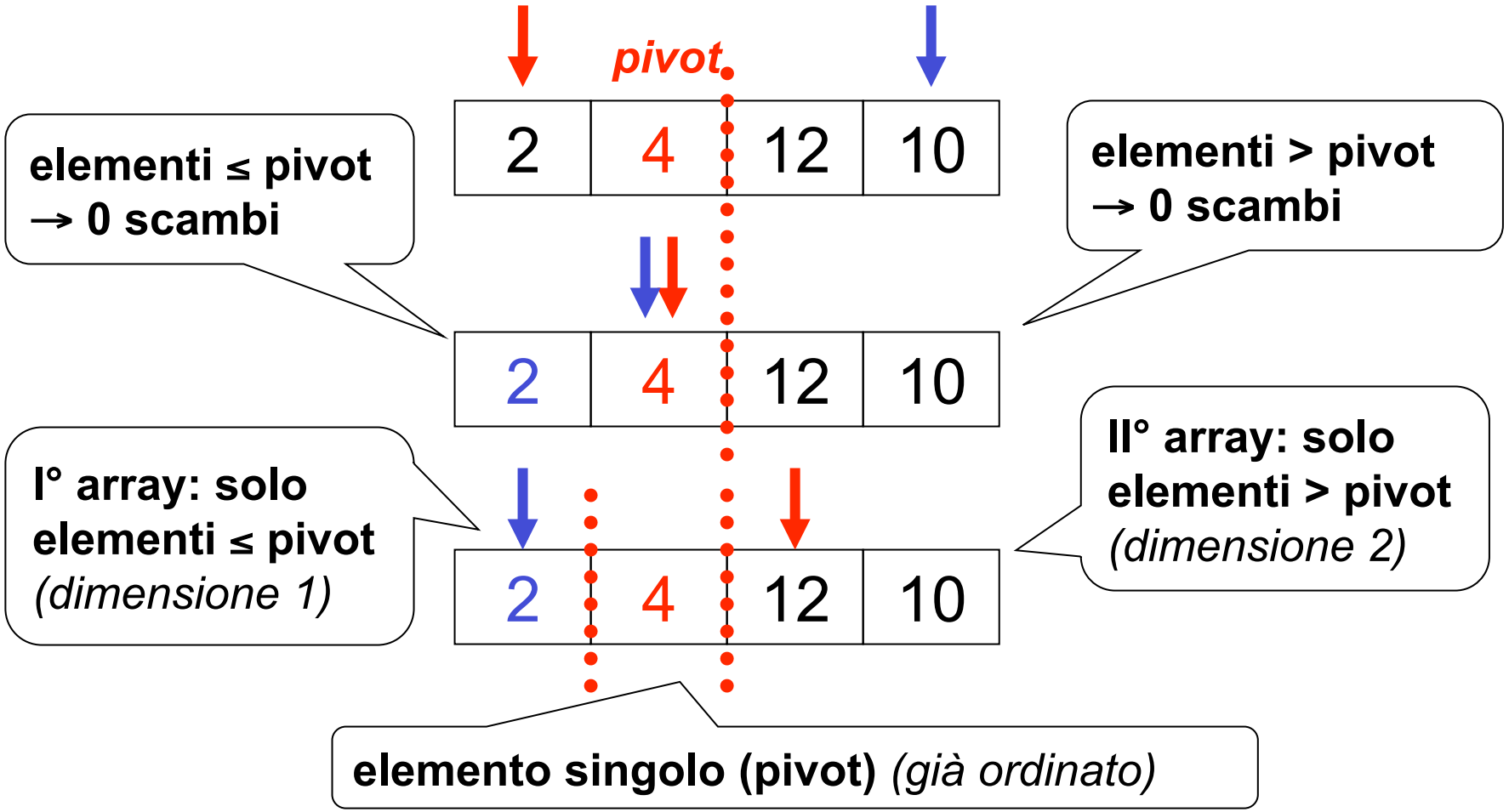
QUICK SORT

Esempio (ipotesi: si sceglie 14 come pivot)



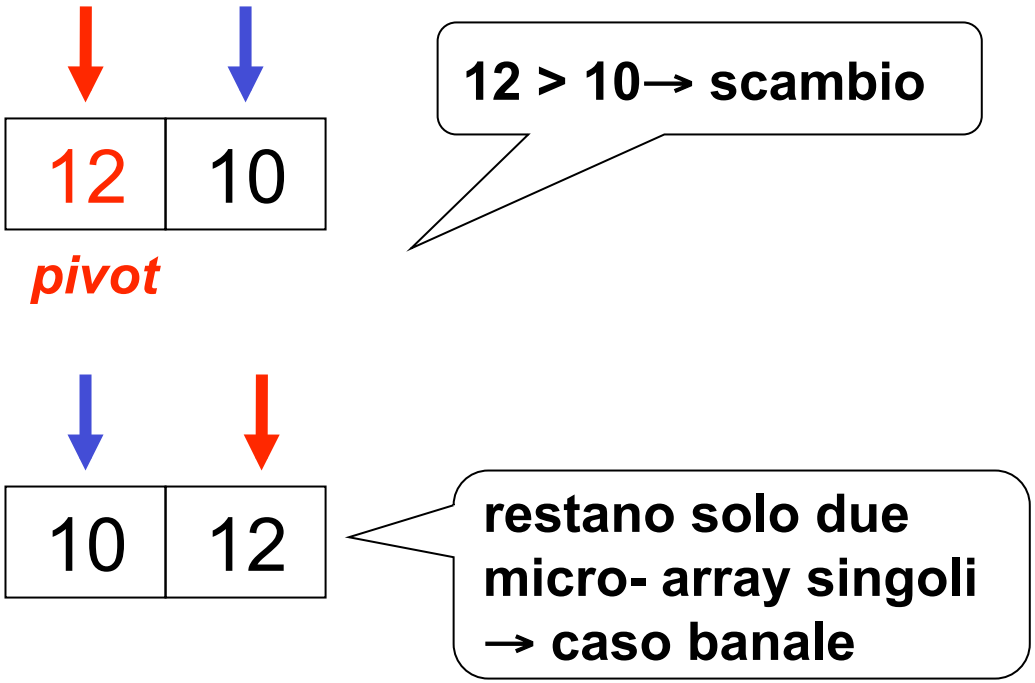
QUICK SORT

Esempio (passo 2: ricorsione sul I° sub-array)



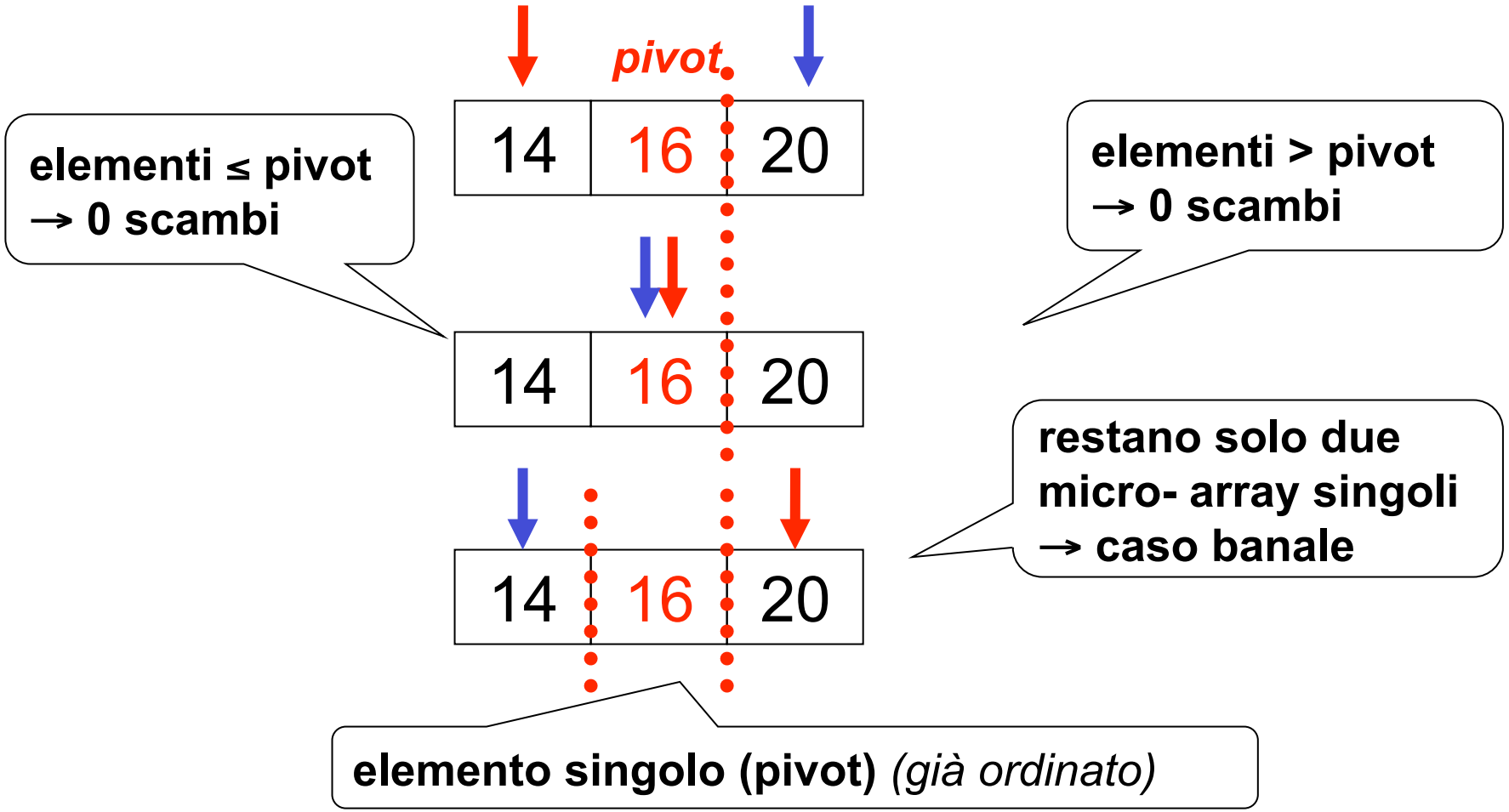
QUICK SORT

Esempio (passo 3: ricors. sul II° sub-sub-array)



QUICK SORT

Esempio (passo 4: ricorsione sul II° sub-array)



QUICK SORT

Specifica

```
void quickSort(int v[],int iniz,int fine){
 if (<vettore non vuoto> )
   <scegli come pivot l'elemento mediano>
   <isola nella prima metà array gli elementi minori o
    uguali al pivot e nella seconda metà quelli maggiori >
   <richiama quicksort ricorsivamente sui due sub-array,
    se non sono vuoti >
}
```

QUICK SORT

Codifica

```
void quickSort(int v[],int iniz,int fine){
 int i, j, pivot;
 if (iniz<fine) {
   i = iniz,  j = fine;
   pivot = v[(iniz + fine)/2];
```

<isola nella prima metà array gli elementi minori o uguali al pivot e nella seconda metà quelli maggiori >

<richiama quicksort ricorsivamente sui due sub-array, se non sono vuoti >

```
}
```

QUICK SORT

Codifica

```
void quickSort(int v[],int iniz,int fine){
 int i, j, pivot;
 if (iniz<fine) {
   i = iniz,  j = fine;
   pivot = v[(iniz + fine)/2];
```

<isola nella prima metà array gli elementi minori o uguali al pivot e nella seconda metà quelli maggiori >

```
   if (iniz < j) quickSort(v, iniz, j);
   if (i < fine) quickSort(v, i, fine);
}
```

QUICK SORT

Codifica

<isola nella prima metà array gli elementi minori o uguali al pivot e nella seconda metà quelli maggiori >

```
do {
      while (v[i] < pivot) i++;
      while (v[j] > pivot) j--;
      if (i < j) scambia(&v[i], &v[j]);
      if (i <= j) i++, j--;
} while (i <= j);
```

<invariante: qui j<i, quindi i due sub-array su cui applicare la ricorsione sono (iniz,j) e (i,fine) >

QUICK SORT

La complessità dipende dalla scelta del pivot:

- se il pivot è scelto male (uno dei due sub-array ha lunghezza zero), i confronti sono $O(N^2)$
- **se però il pivot è scelto bene (in modo da avere due sub-array di egual dimensione):**
  - **si hanno $\log_2 N$ attivazioni di quicksort**
  - **al passo k si opera su 2^k array, ciascuno di lunghezza $L = N/2^k$**
  - **il numero di confronti ad ogni livello è sempre N (L confronti per ciascuno dei 2^k array)**
- **Numero globale di confronti: $O(N \log_2 N)$**

QUICK SORT

- **Si può dimostrare che $O(N \log_2 N)$ è un limite inferiore alla complessità del *problema dell'ordinamento di un array.***
- **Dunque, *nessun algoritmo, presente o futuro, potrà far meglio di* $O(N \log_2 N)$**
- **Però, il quicksort raggiunge questo risultato *solo se il pivot è scelto bene***
  - **per fortuna, la suddivisione in sub-array uguali è la cosa più probabile nel caso medio**
  - **l'ideale sarebbe però che tale risultato fosse raggiunto *sempre:* a ciò provvede il *Merge Sort.***

ESPERIMENTI

- **Verificare le valutazioni di complessità che abbiamo dato non è difficile**
  - **basta predisporre un programma che "conti" le istruzioni di confronto, incrementando ogni volta un'apposita variabile intera ...**
  - **... e farlo funzionare con diverse quantità di dati di ingresso**
- **Farlo può essere molto significativo.**

ESPERIMENTI

- **Risultati:**

N	$N^2/2$	$N^2/4$	$N \log_2 N$	naive sort	bubble sort	insert sort	quick sort	merge sort
15	112	56	59	119	14	31	57	39
45	1012	506	247	1034	900	444	234	191
90	4050	2025	584	4094	2294	1876	555	471
135	9112	4556	955	9179	3689	4296	822	793

- per problemi semplici, anche gli algoritmi “poco sofisticati” funzionano abbastanza bene, *a volte meglio degli altri*
- quando invece il problema si fa complesso, la differenza diventa ben evidente.