COMPLESSITA' COMPUTAZIONALE

(II PARTE)

Tutor: Francesca Piersigilli

Teoria della complessità

- Pone le basi per stabilire:
  1. la complessità di un problema
  2. l’efficienza di un algoritmo
     - quanta memoria usa?
     - quanto tempo impiega?
- Offre una descrizione quantitativa ma approssimata di qualunque grandezza di interesse:
  1. numero di confronti eseguiti
  2. numero di invocazioni effettuate
  3. numero di somme e prodotti eseguiti
- Cresce al crescere della dimensione della struttura dati in input:
  1. lunghezza di un array
  2. numero di nodi in un albero…

Classi di complessità (I)

In ordine di complessità crescente

- Logaritmica: $O(n^k \log_h n)$ (per qualche k,h)
- Lineare: $O(n)$
- Polinomiale: $O(n^k)$ (per qualche $k>1$)
- Esponenziale: $O(k^n)$
- Fattoriale: $O(n!)$, $O(n^n)$

I problemi con complessità <u>al più</u> polinomiale si definiscono ***trattabili;***

I problemi con complessità <u>almeno</u> esponenziale si definiscono ***intrattabili***

Classi di complessità (II)

n	**$\log_2 n$**	**$n*\log_2 n$**	**n^2**	**n^3**	**2^n**
2	1	2	4	8	4
10	3,322	33,22	10^2	10^3	$> 10^3$
10^2	6,644	664,4	10^4	10^6	$>> 10^{25}$
10^3	9,966	9996,0	10^6	10^9	$>> 10^{250}$
10^4	13,287	1328,7	10^8	10^{12}	$>> 10^{2500}$

Esempi

- Il numero di confronti medi per cercare un elemento in un array di dimensione **n** è **O(n)**
  - esiste un **algoritmo lineare** che risolve il problema
- Il numero di confronti medi per cercare un elemento in un array ordinato è $\mathbf{O(\log_2 n)}$
  - esiste un **algoritmo logaritmico** che lo risolve

- I precedenti sono tutti problemi trattabili!

Dipendenza dai dati di ingresso

- Spesso accade che il costo di un algoritmo dipenda non solo dalla dimensione dei dati di ingresso, ma anche **dai loro particolari valori**
  - ad esempio, un algoritmo che ordina un array può avere un costo diverso a seconda se l’array è “molto disordinato” o invece “quasi del tutto ordinato”
  - analogamente un algoritmo che ricerca un elemento in un array può costare poco, se l’elemento viene trovato subito, o molto di più, se l’elemento si trova “in fondo” o è magari del tutto assente.

Dipendenza dai dati di ingresso

In queste situazioni occorre distinguere diversi casi (migliore, peggiore, medio):

<u>ESEMPIO:</u>

Per la **ricerca sequenziale** in un array, il costo dipende dalla posizione dell'elemento cercato.

- ***Caso migliore***: l'elemento è il primo dell'array $\longrightarrow$ un solo confronto;
- ***Caso peggiore***: l'elemento è l'ultimo o non è presente $\longrightarrow$ **n** confronti, costo lineare **O(n)**;
- ***Caso medio***: l'elemento può, con egual probabilità, essere il primo (1 confronto), il secondo (2 confronti), … o l'ultimo (**n confronti**):

$$\sum \mathrm{Prob}(el(i)) * i \;=\; \sum (1/N) * i \;=\; (N+1)/2 = O(N/2)$$

Algoritmi di ordinamento

- **Scopo**: ordinare una sequenza di elementi in base a una certa relazione d'ordine:

  - Lo scopo finale è ben definito:
    $\longrightarrow$ *algoritmi equivalenti*

  - Diversi algoritmi possono avere
    $\longrightarrow$ *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)

•**quick sort** (non intuitivo, alquanto efficiente)

•**merge sort** (non intuitivo, molto efficiente)

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

Naïve sort

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

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] >
        // invariante: v[n-1] contiene il massimo
        < restringi l'attenzione alle prime n-1 caselle dell'array,
        ponendo n' = n-1>
}
```

Naïve sort

<u>Codifica</u>

```
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 ad ogni iterazione

Naïve sort

<u>Codifica</u>

```
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

<u>Valutazione di complessità:</u>

Il numero di confronti necessari vale sempre:

$$(N-1) + (N-2) + (N-3) + \ldots + 2 + 1 =$$
$$= N*(N-1)/2 \quad = \quad O(N^2/2)$$

•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, che per un array già ordinato!!

Bubble sort

- Corregge il difetto principale del naive sort: quello di non accorgersi se l’array, a un certo punto, è già ordinato.

- Opera per “passate successive” sull’array:
    - Ad 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.

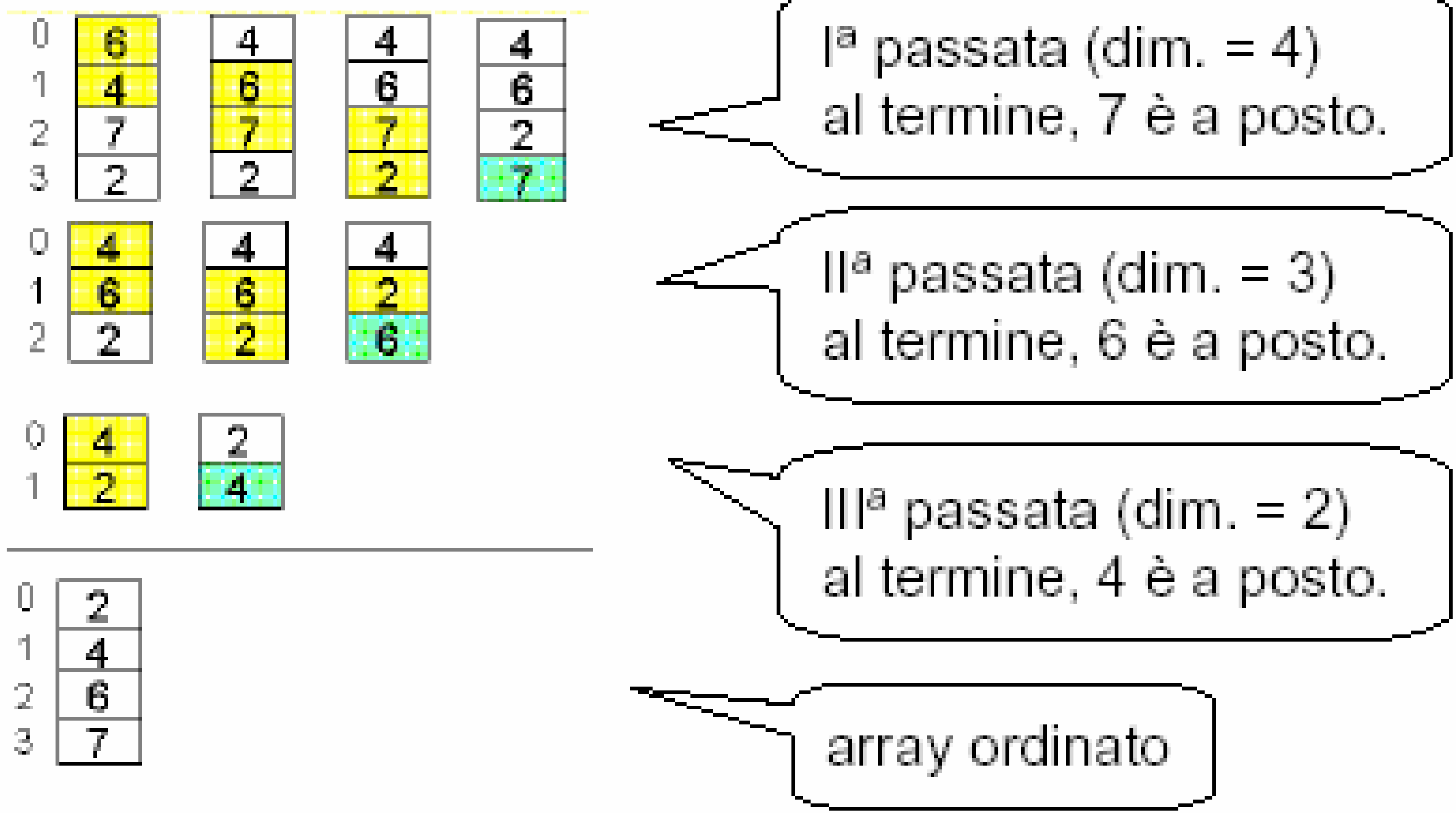
Bubble sort

Codifica

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

Bubble sort

Esempio



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*.

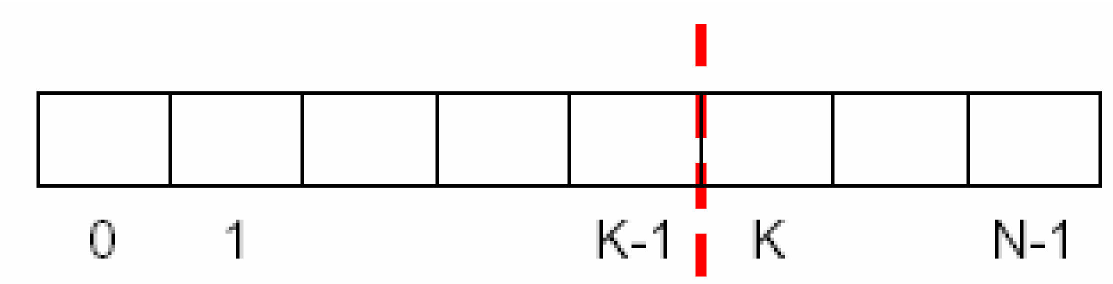
• Idealmente, il metodo costruisce un nuovo array, contenente gli stessi elementi del primo, ma ordinato.

• 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

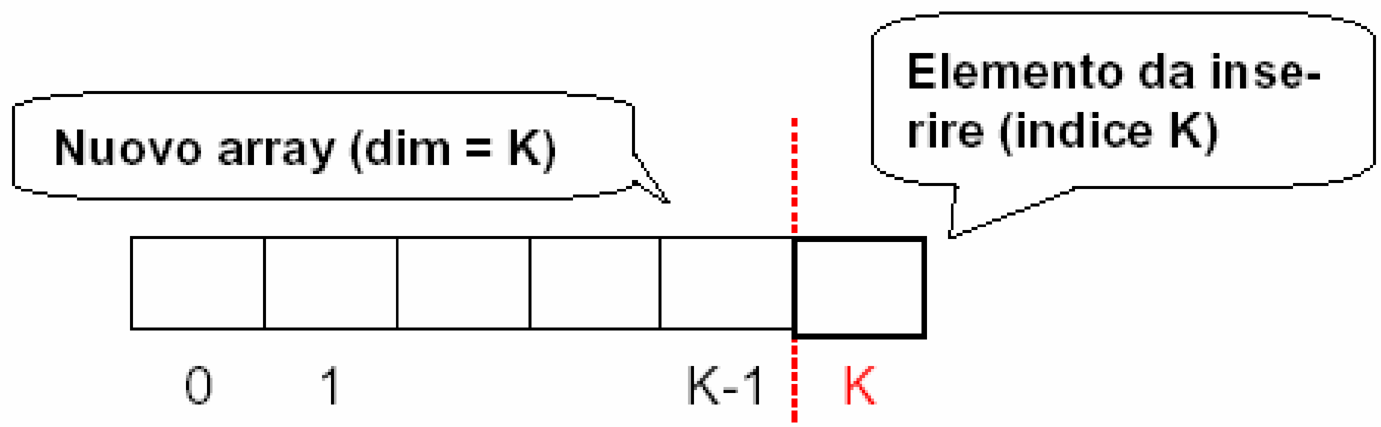
Scelta di progetto

- "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

<u>*Specifica*</u>

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

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

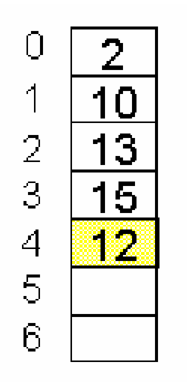
<u>*Codifica*</u>

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

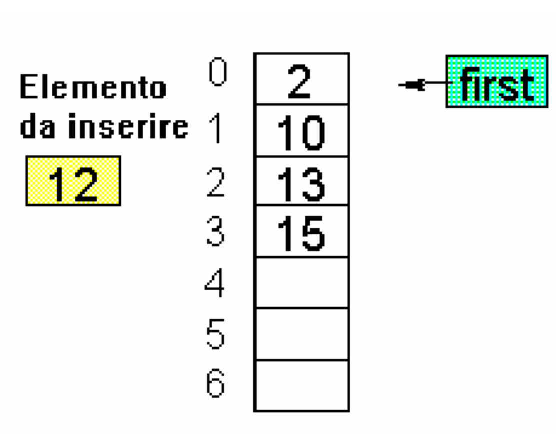
Codifica: **All'inizio (k=1) il nuovo array è la sola prima cella**

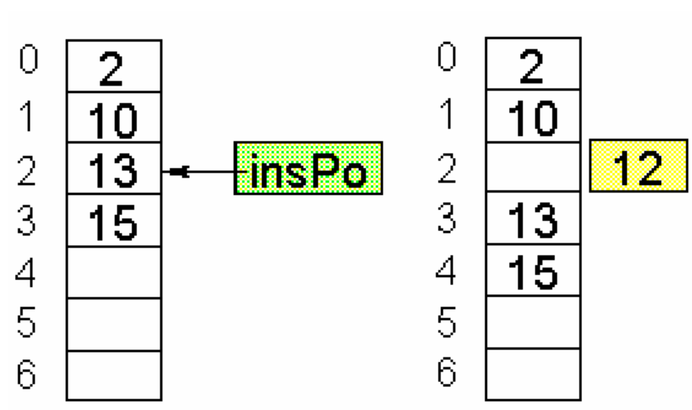
Insert sort

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

<u>*Specifica di insMinore()*</u>

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>

<inserisce il nuovo elemento nella posizione prevista>

}

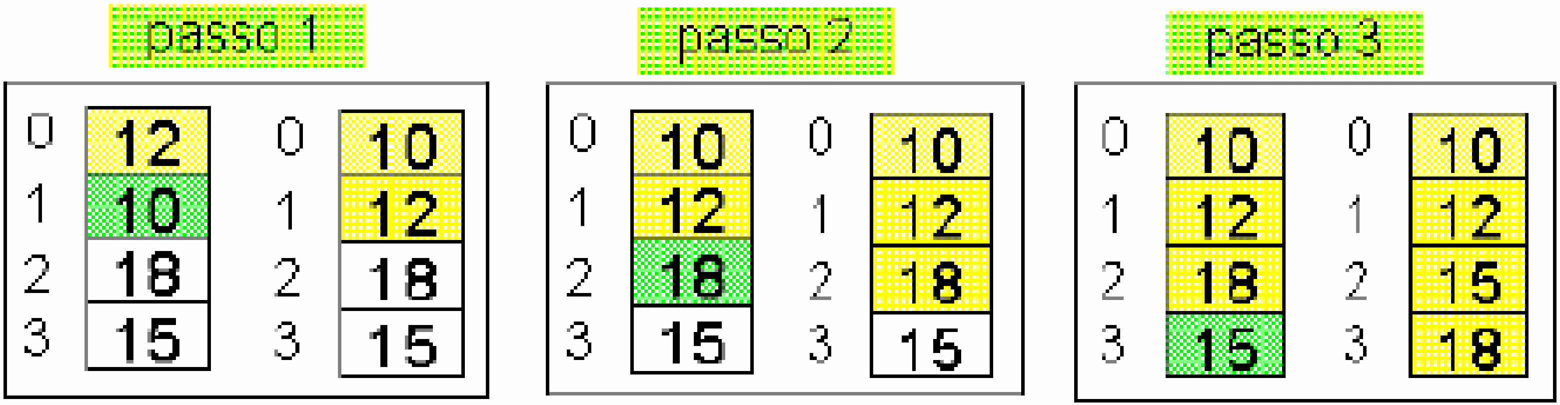
Insert sort

<u>*Codifica di insMinore()*</u>

```
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



passo 1
0 12 0 10
1 10 1 12
2 18 2 18
3 15 3 15
passo 2
0 10 0 10
1 12 1 12
2 18 2 18
3 15 3 15
passo 3
0 10 0 10
1 12 1 12
2 18 2 15
3 15 3 18

Insert sort

Valutazione di complessità

• *Nel caso peggiore* (array ordinato al contrario), richiede $1+2+3+\ldots+(N-1)$ confronti e spostamenti → **$O(n^2/2)$**

• *Nel caso migliore* (array già ordinato), bastano solo n-1 confronti (senza spostamenti)

• ***Nel caso medio*****,** ad ogni ciclo il nuovo elemento viene inserito nella posizione centrale dell'array → $1/2+2/2+\ldots+(n-1)/2$ confronti e spostamenti.

Quindi: $O(n^2/4)$

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 *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 ad un array di un solo elemento, che è già ordinato (*caso banale*).

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*

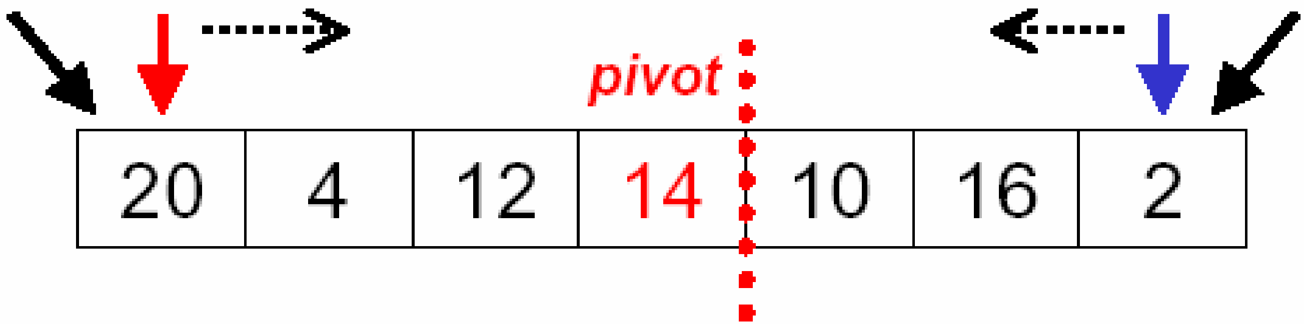
Poi si riapplica quicksort ai due sub-array.

Quick sort

<u>Esempio: legenda</u>

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

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

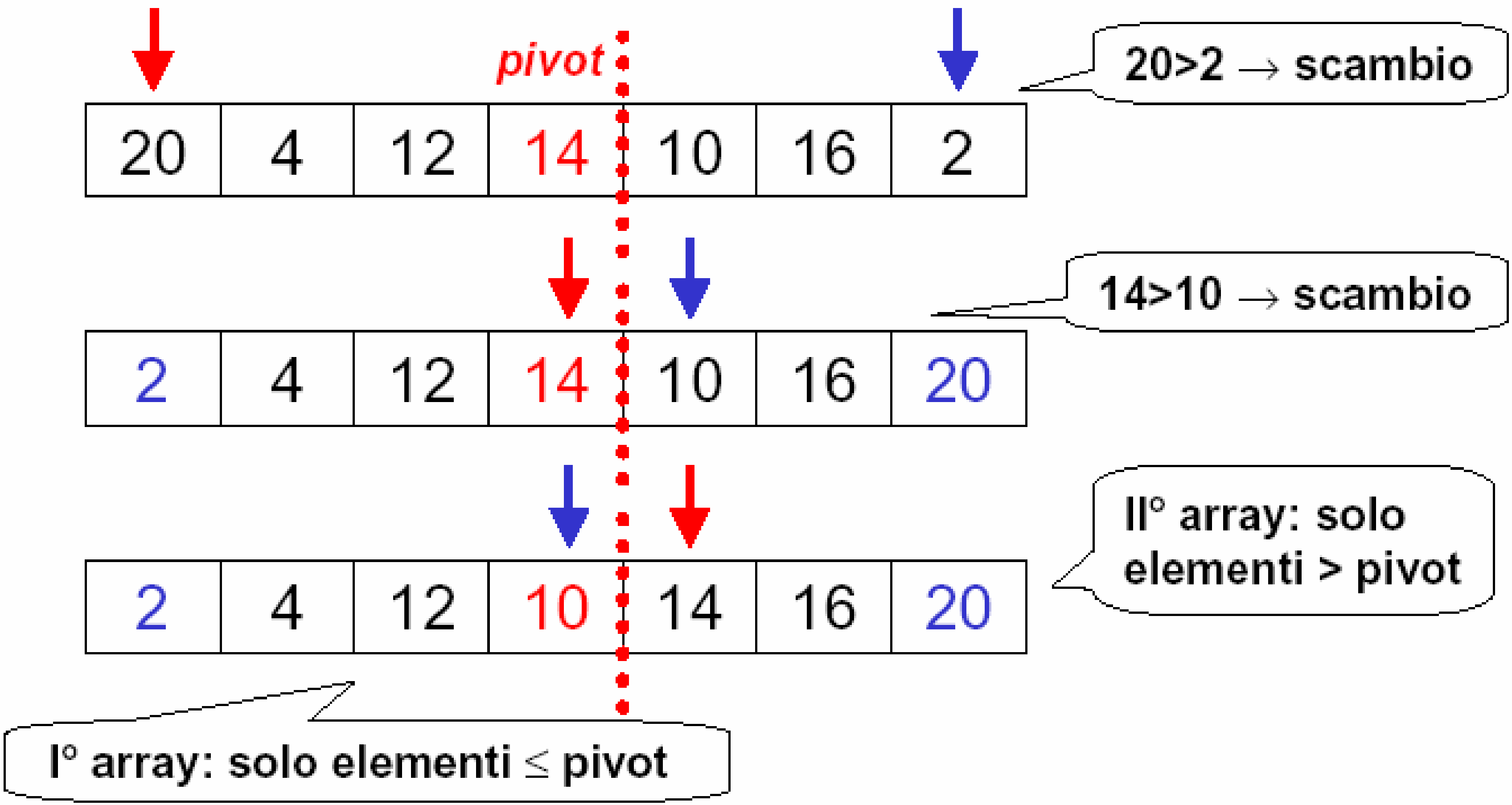


freccia nera (iniz): indica l'inizio dell'array (e del I° sub-array)

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

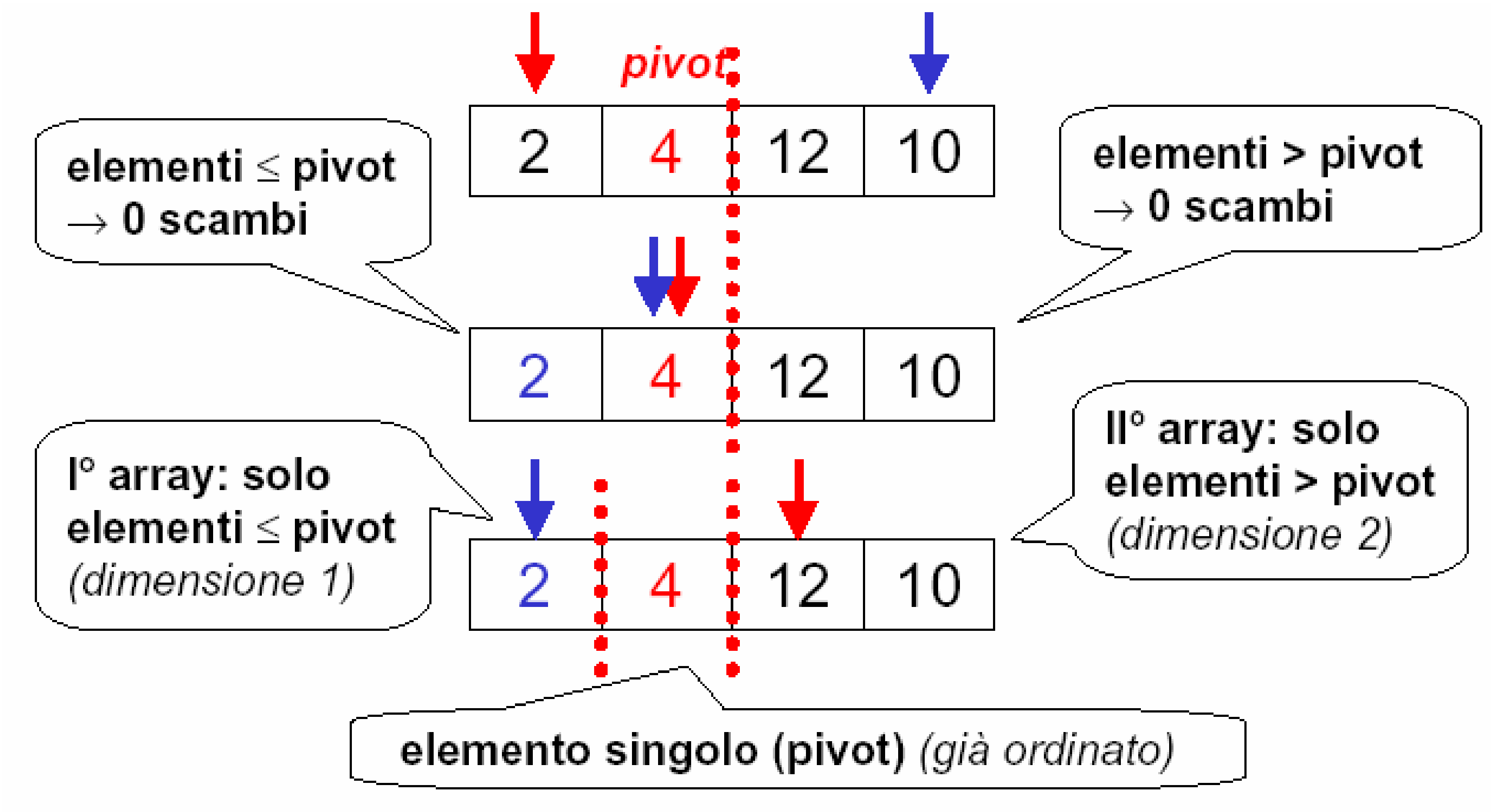
Quick sort

Esempio



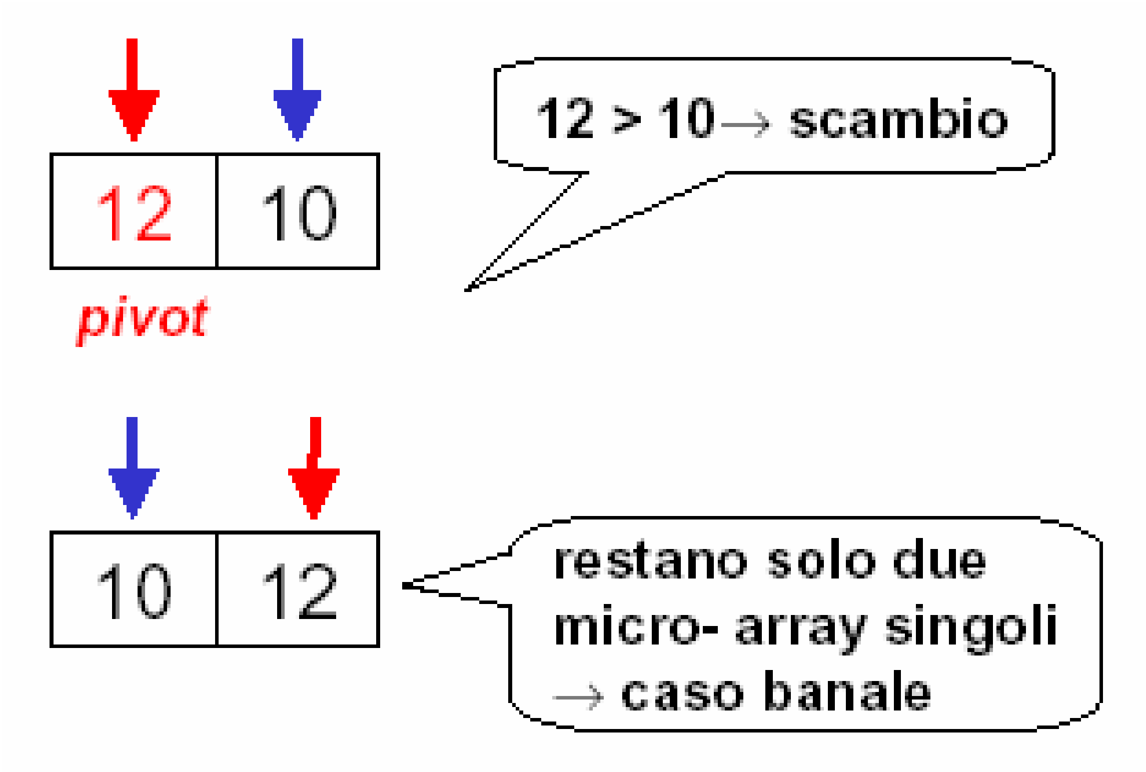
Quick sort

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



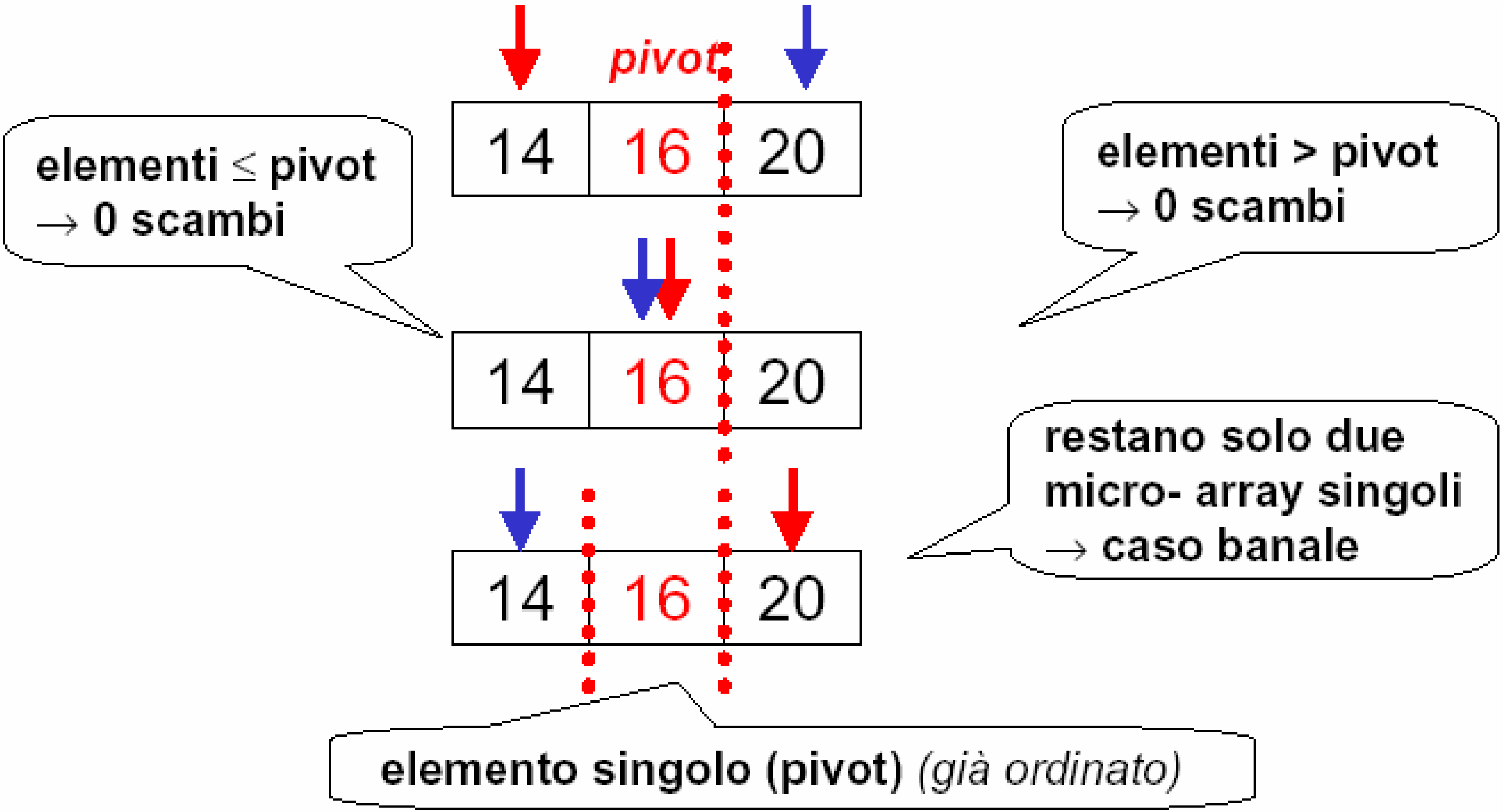
Quick sort

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



Quick sort

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



Quick sort

<u>Specifica</u>

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 dell'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 dell'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 $\mathbf{O(n^2)}$

• se però il pivot è scelto bene (in modo da avere due sub-array di egual dimensione):

- si hanno $\mathbf{\log_2 n}$ 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: $\mathbf{O(n \log_2 n)}$

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***.

Merge sort

• È una variante del quick sort che produce *sempre* due sub-array di egual ampiezza

– così, si ottiene sempre il caso ottimo **O(n $\log_2$ n)**

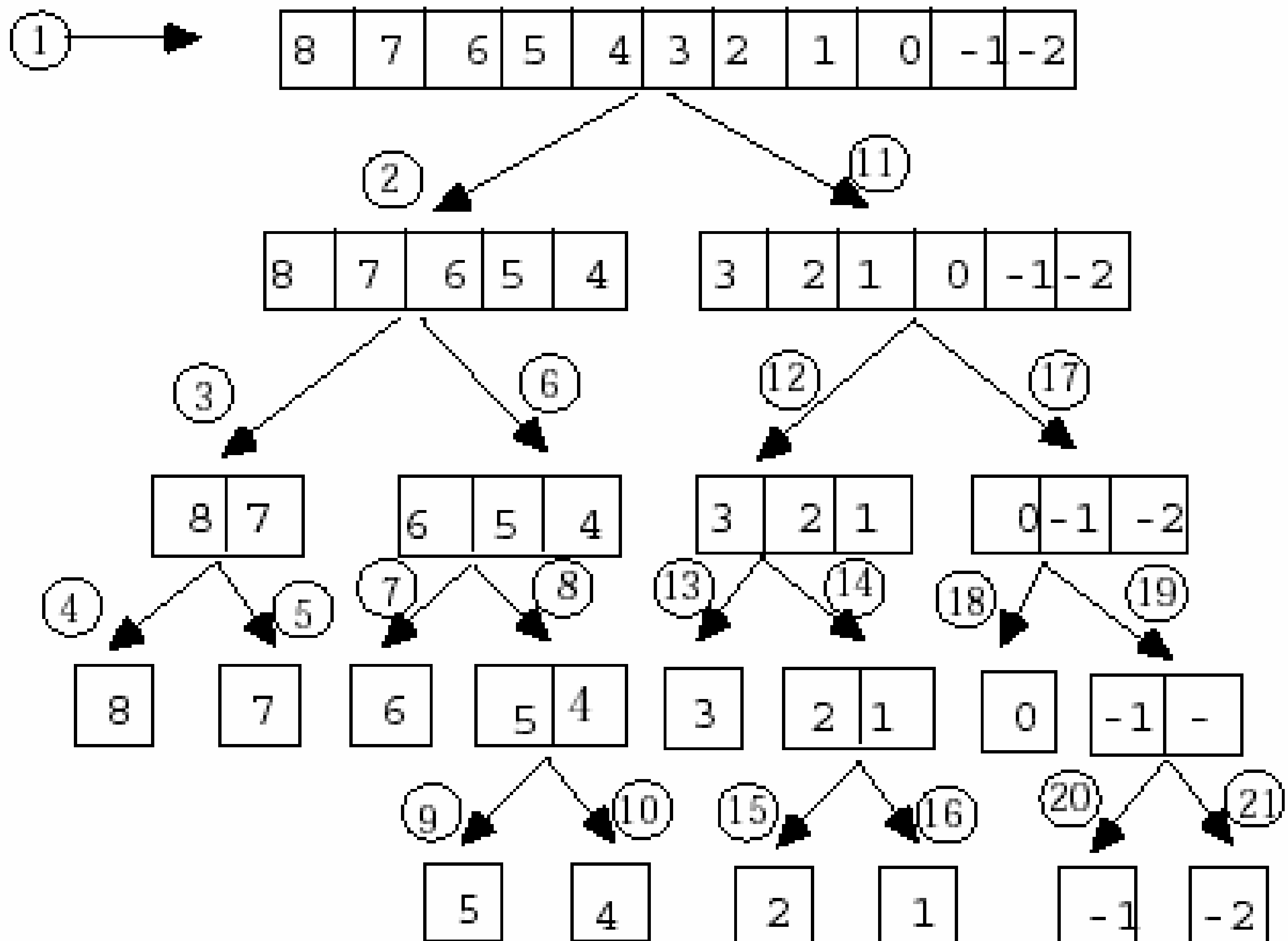
• *In pratica:*

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



Merge sort

Specifica

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

Merge sort

Codifica

```
void mergeSort(int v[], int first, int last, int vuot[]) {
        int mid;
        if ( first < last ) {
                mid = (last + first) / 2;
                mergeSort(v, first, mid, vuot);
                mergeSort(v, mid+1, last, vuot);
                merge(v, first, mid+1, last, vuot);
        }
}
```

mergeSort() si limita a suddividere l’array:è

merge() che svolge il lavoro

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

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.