Ordinamento

Carla Binucci e Walter Didimo

Il problema dell'ordinamento

In generale, il Problema dell'ordinamento consiste nel trasformare una sequenza di elementi rendendola ordinata rispetto a un certo criterio

- Il vantaggio principale nel mantenere ordinata una collezione di dati consiste nel rendere più efficienti le operazioni di ricerca
  - ad esempio, la ricerca in un array ha in generale costo lineare, ma se l'array è ordinato allora può essere eseguita con costo logaritmico

Vogliamo studiare il problema dell'ordinamento di un array e alcuni algoritmi per la sua soluzione, di cui valuteremo la complessità

Il problema dell'ordinamento di un array

Il problema dell'ordinamento non decrescente di un array di interi è definito come segue:

- sia *dati* un array di elementi di tipo *int*
- trasformare l'array *dati* in modo tale che i suoi elementi siano in ordine non decrescente
  - ovvero, modificare la posizione degli elementi di *dati* in modo tale che, per ogni indice i, l'elemento dati[i] sia non minore di tutti gli elementi di dati di indice j, con j<i

Se gli elementi dell'array sono tutti distinti, l'ordinamento non decrescente coincide con l'ordinamento crescente

Ordinamento di un array - esempio

Ordinamento di un array di interi in modo non decrescente

- esempio di sequenza non ordinata

10	2	16	0	-1	51	4	23	9	8

$> \quad \leq \quad > \quad > \quad \leq \quad > \quad \leq \quad > \quad \leq$

- la stessa sequenza, ordinata in modo non decrescente

-1	0	2	4	8	9	10	16	23	51

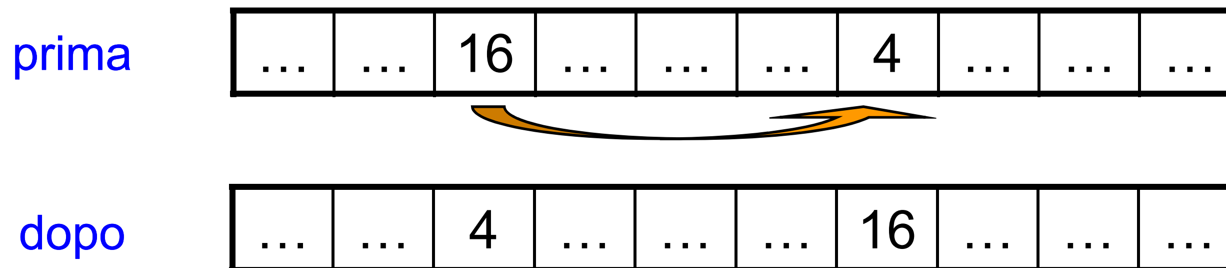
$\leq \quad \leq \quad \leq \quad \leq \quad \leq \quad \leq \quad \leq \quad \leq \quad \leq$

- essendo gli elementi tutti distinti, allora l'array è ordinato anche in modo crescente

Algoritmi di ordinamento e scambi

Gli algoritmi di ordinamento che esaminiamo sono basati sulla seguente strategia generale:

- confronta gli elementi dell'array a coppie e, se gli elementi di una coppia non sono ordinati tra di loro, scambiali



- lo scambio di elementi che non sono ordinati tra di loro aumenta il "grado di ordinamento" dell'array

- quando non ci sono più coppie di elementi che sono non ordinate, allora l'intero array è ordinato

I diversi algoritmi di ordinamento si differenziano in base a come vengono selezionate (e scambiate) le coppie di elementi da confrontare

Scambio di elementi in un array

Ecco il metodo che usiamo per scambiare (to swap) una coppia di elementi in un array di interi

```
/* scambia gli elementi di indice i e j di dati */
private static void scambia(int[] dati, int i, int j){
    // pre: dati!=null && 0<= i,j <dati.length
    int temp;
    temp = dati[i];
    dati[i] = dati[j];
    dati[j] = temp;
}
```

Elementi ordinati e elementi non ordinati

Molti algoritmi di ordinamento partizionano gli elementi dell'array in due insiemi (ciascuno dei quali è composto da elementi contigui)

- l'insieme degli elementi ordinati che è formato da elementi che sono stati collocati nella loro posizione definitiva
  - l'algoritmo di ordinamento non deve più confrontare né scambiare questi elementi
- l'insieme degli elementi non ordinati che è formato da elementi che non sono ancora stati collocati nella loro posizione definitiva
  - l'algoritmo di ordinamento deve confrontare ed eventualmente scambiare solo questi elementi

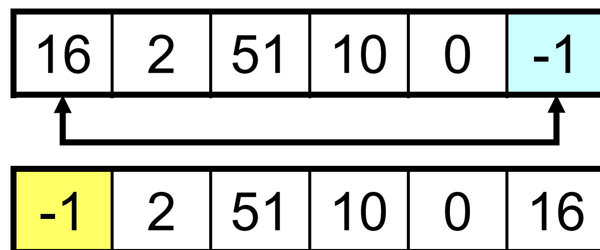
Ordinamento per selezione

Il primo algoritmo di ordinamento che studiamo è: l'ordinamento per selezione (selection sort) che implementa la seguente strategia:

- fino a quando l'array è non ordinato
  - seleziona l'elemento di valore minimo dell'array tra quelli che non sono stati ancora ordinati
  - disponi l'elemento di valore minimo nella sua posizione definitiva

Una passata del selection sort

L'algoritmo procede per fasi, che sono chiamate passate:



Ad ogni *passata*, l'algoritmo

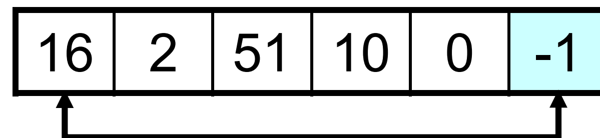
- seleziona l'elemento di valore minimo (tra quelli non ordinati) e la sua posizione (nell'esempio, l'elemento di valore –1 in posizione 5)

- colloca, mediante uno scambio, tale elemento nella sua posizione definitiva, cioè la posizione più a sinistra tra gli elementi non ordinati (nell'esempio, in posizione 0)

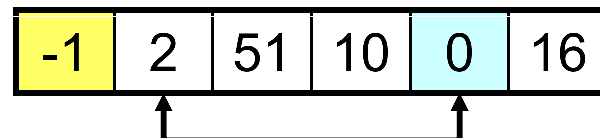
Effetto di una passata del selection sort

Nell'ordinamento per selezione inizialmente tutti gli elementi sono non ordinati:

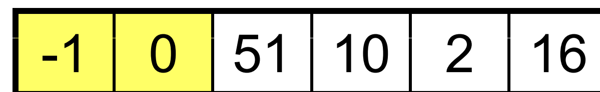
- dopo la *prima passata* il primo elemento viene ordinato
- dopo la *seconda passata* il secondo elemento viene ordinato
- …..



prima passata



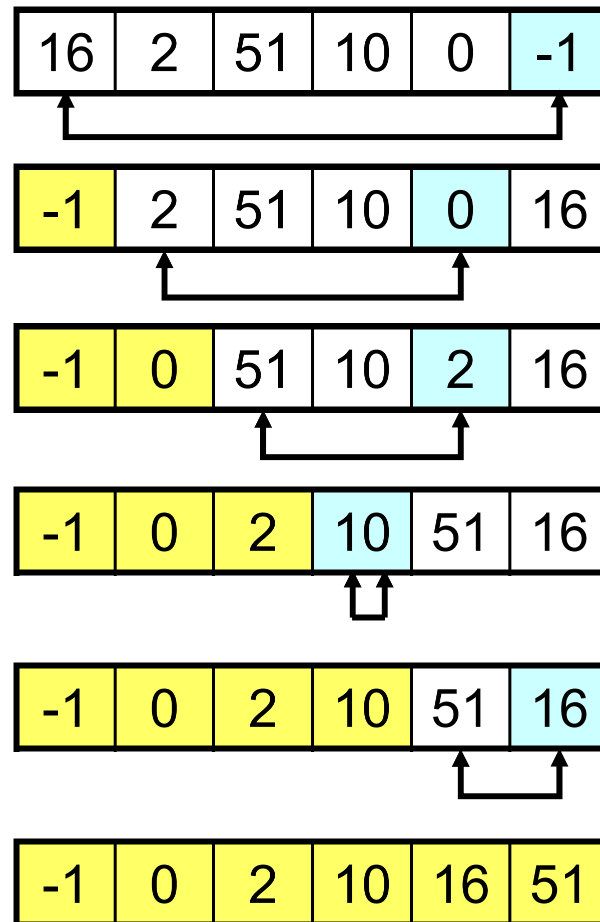
seconda passata



N-1 passate ordinano l'intero array (nota che al termine della passata i l'elemento minimo viene messo in posizione i-1)

N-1 passate del selection sort

N-1 passate dell'algoritmo ordinano un array di *N* elementi



Implementazione del selection sort

```
/* Ordinamento per selezione */
public static void selectionSort(int[] dati) {
     int n = dati.length;    // numero di elementi
     int i;                  // per la scansione di dati
     int ordinati;           // num. di elementi ordinati
     int imin;      // indice del min. tra i non ordinati
     ordinati = 0;
     while (ordinati<n) {
         /* cerca il minimo tra i non ordinati */
         imin = ordinati;
         for (i=ordinati+1; i<n; i++)
             if (dati[i]<dati[imin])
                 imin = i;
         /* l'elemento in imin viene ordinato */
         scambia(dati, ordinati, imin);
         ordinati++;
        }
}
```

Complessità del selection sort

Qual è l' *operazione dominante* per il selection sort?

- l'operazione dominante è il confronto tra coppie di elementi (dati[i]<dati[imin])
  - è una delle operazioni più annidate
  - l'operazione più annidata è l'assegnazione imin = i, ma ha lo stesso costo del confronto tra elementi e viene eseguita meno spesso del confronto tra elementi

Quante volte viene eseguita l'operazione dominante nel caso peggiore?

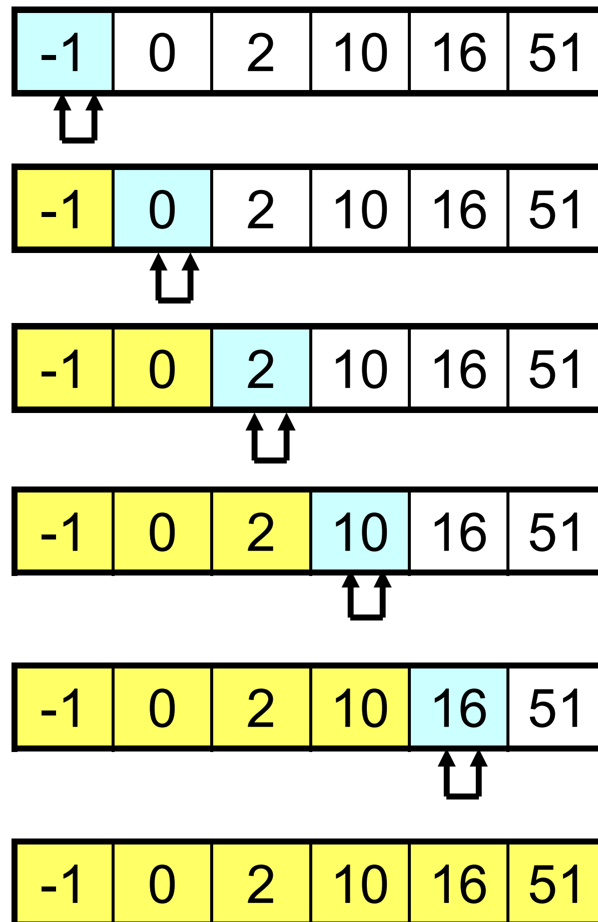
Complessità del selection sort

Qual è il *caso peggiore* per il selection sort?

- è facile verificare che il comportamento dell'ordinamento per selezione è indipendente dall'ordine preesistente nell'array da ordinare
  - il caso migliore e il caso peggiore coincidono
  - per N–1 volte bisogna comunque determinare il minimo tra gli elementi non ordinati ed effettuare uno scambio

Passate su un array ordinato

Il selection sort esegue *N-1* passate anche se l'array è ordinato



Il selection sort ha complessità quadratica

Esecuzione dell' *operazione dominante* per il selection sort

- durante la prima passata vengono eseguiti N-1 confronti per determinare il minimo tra N elementi
- durante la seconda passata vengono eseguiti N-2 confronti per determinare il minimo tra N-1 elementi
- …..
- durante la N-1-esima passata viene eseguito 1 confronto per determinare il minimo tra 2 elementi

Il numero di confronti è $\sum_{i=1}^{N-1} (N-i) = N\,(N-1)/2$

Quindi, il selection sort ha complessità asintotica *$O(N^2)$* - cioè quadratica

Ordinamento per inserzione

L'algoritmo di ordinamento per inserzione (insertion sort) implementa la seguente strategia:

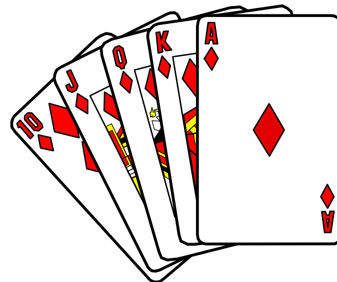
- gli elementi sono partizionati in due insiemi
  - uno di elementi relativamente ordinati memorizzati nelle posizioni più a sinistra dell'array (sono elementi ordinati tra di loro, ma che non sono stati necessariamente collocati nelle loro posizioni definitive)
  - uno di elementi non relativamente ordinati memorizzati nelle posizioni più a destra dell'array

Ordinamento per inserzione

L'algoritmo di ordinamento per inserzione (insertion sort) implementa la seguente strategia:

- ad ogni *passata*
  - il primo tra gli elementi non relativamente ordinati viene collocato tra gli elementi relativamente ordinati, inserendolo nella posizione corretta (relativamente agli altri elementi relativamente ordinati)

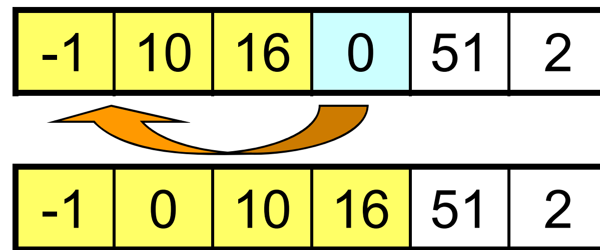
L'insertion sort è solitamente utilizzato dai giocatori di carte



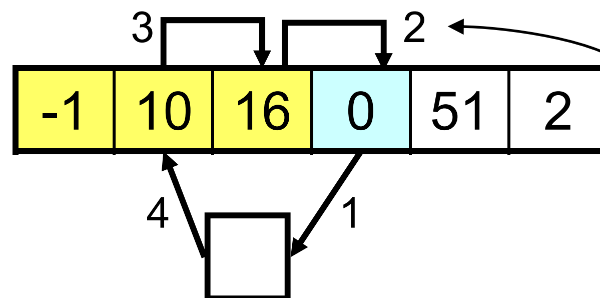
Una passata dell'insertion sort

Ecco l'effetto di una passata dell'insertion sort:

- supponiamo che i primi tre elementi dell'array siano quelli relativamente ordinati e che lo zero debba essere ordinato



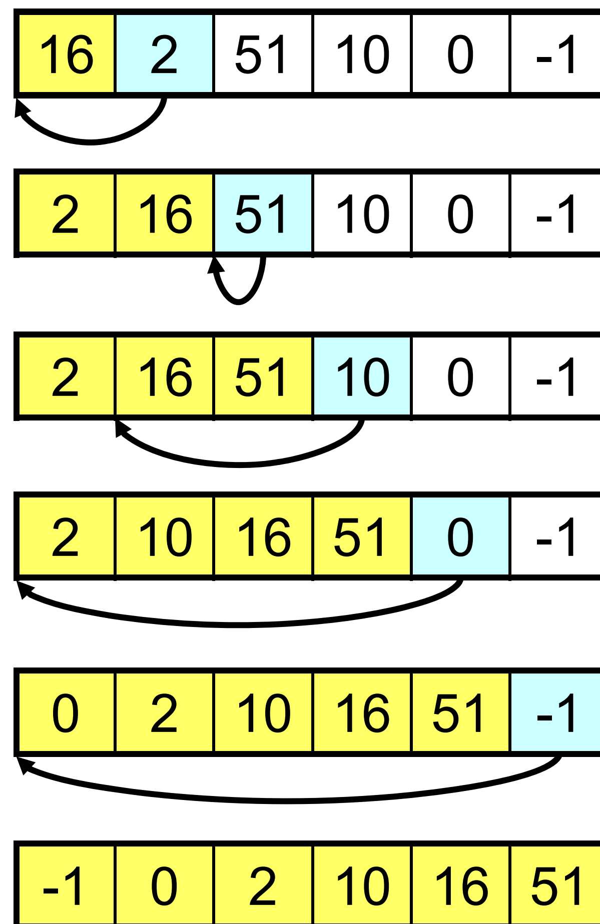
- la *passata* inserisce lo 0 dopo il -1, gli elementi 10 e 16 vengono spostati verso destra



i numeri indicano l'ordine in cui devono essere eseguite le varie operazioni

N-1 passate dell'insertion sort

N-1 passate dell'algoritmo ordinano un array di *N* elementi



Implementazione dell’insertion sort

```
/* Ordinamento per inserzione */
  public static void insertionSort(int[] dati) {
        int n = dati.length;  // numero di elementi
        int i;                // per la scansione di dati
        int ordinati;         // num. di elementi "ordinati"
        int corrente;         // elemento da "ordinare"
        boolean inseribile;   // è possibile inserire
                              // corrente tra gli "ordinati"

        ... da completare ...

    }
```

Implementazione dell'insertion sort

Ecco la porzione di codice mancante

```
ordinati = 1;
    while (ordinati<n) {
        /* considera il primo tra i "non ordinati" */
        corrente = dati[ordinati];
        /* inserisce corrente tra gli "ordinati" */
        inseribile = false;
        i = ordinati;
        while (!inseribile && i>0)
            if (corrente < dati[i-1]) {
                dati[i] = dati[i-1];
                i--;
            } else
                inseribile = true;
        // re-inserisce corrente
        dati[i] = corrente;  // lo inserisce
        ordinati++;
    }
```

Complessità dell'insertion sort

Qual è l' *operazione dominante* per l'insertion sort?

- l'operazione dominante è il confronto tra coppie di elementi (corrente<dati[i])

Qual è il *caso peggiore* per l'insertion sort?

- il caso peggiore si verifica quando l'array è ordinato in modo non crescente

Quante volte viene eseguita l'operazione dominante nel caso peggiore?

L'insertion sort ha complessità quadratica

Esecuzione dell' operazione dominante nel caso peggiore

- durante la prima passata viene eseguito 1 confronto
- durante la seconda passata vengono eseguiti 2 confronti
- …..
- durante la N-1-esima passata vengono eseguiti N-1 confronti

Il numero di confronti è $\sum_{i=1}^{N-1} (N-i) = N (N-1)/2$

Quindi, l'insertion sort ha complessità asintotica *$O(N^2)$* -cioè quadratica

Esercizio: determinare il caso migliore e la relativa complessità asintotica

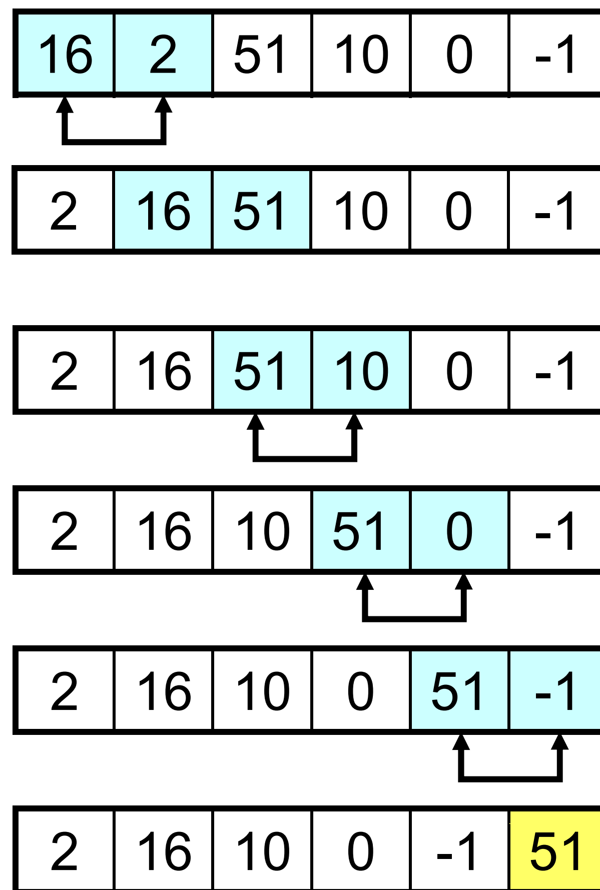
Ordinamento a bolle

L'algoritmo di ordinamento a bolle (bubble sort) implementa la seguente strategia:

- ad ogni *passata*
  - confronta tutte le coppie di elementi adiacenti tra gli elementi non ordinati dell'array
  - ogni volta che una coppia di elementi adiacenti non è ordinata correttamente, scambia gli elementi

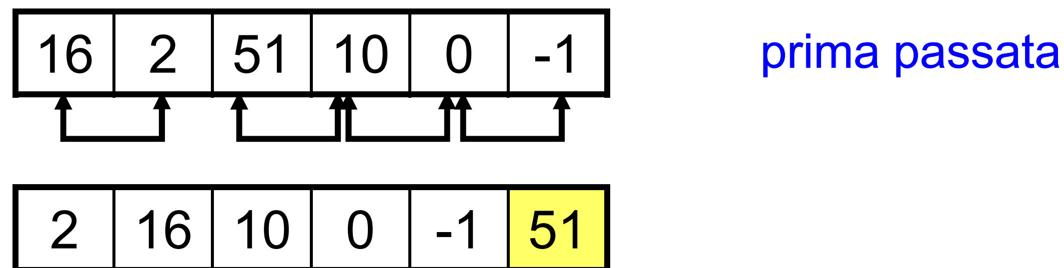
Una passata del bubble sort

Ecco come avviene una passata del bubble sort:



Effetto di una passata del bubble sort

L'effetto di una *passata* dell'ordinamento a bolle è il seguente:

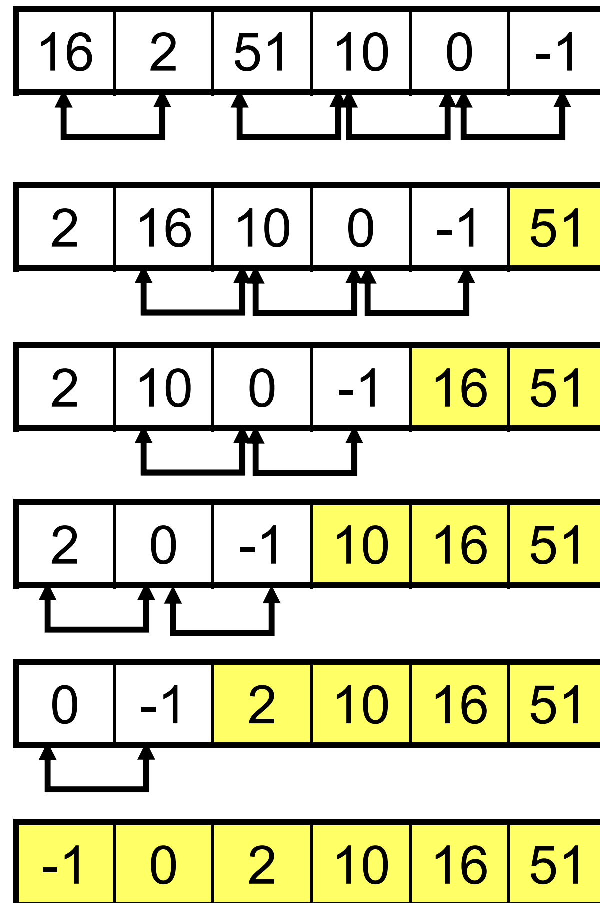


- l'array non è ancora ordinato
- ma l'elemento di valore massimo è stato collocato nella sua posizione definitiva, cioè è stato ordinato
  - la bolla "più grande" è salita fino alla sua posizione definitiva

N-1 passate ordinano l'intero array

N-1 passate del bubble sort

N-1 passate dell'algoritmo ordinano un array di *N* elementi

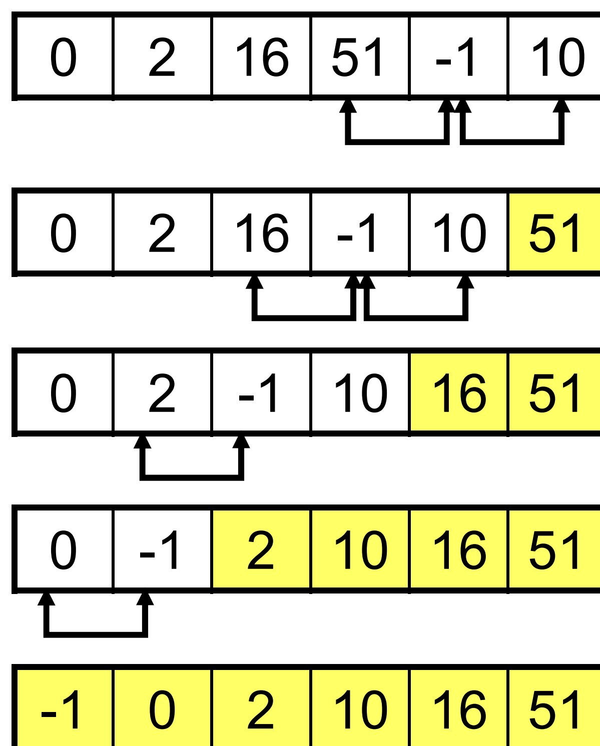


Implementazione elementare del bubble sort

```
/* Ordinamento a bolle - versione elementare */
public static void simpleBubbleSort(int[] dati) {
        // pre: dati!=null
        // post: dati ordinato in modo non decrescente
        int n = dati.length;    // numero di elementi
        int i;                  // per la scansione di dati
        int ordinati;           // num. di elementi ordinati
        ordinati = 0;
        while (ordinati<n) {
            for (i=1; i<n-ordinati; i++)
                if (dati[i]<dati[i-1])
                    scambia(dati, i, i-1);
            // almeno un elemento e' stato ordinato
            ordinati++;
        }
    }
```

Miglioramenti del bubble sort

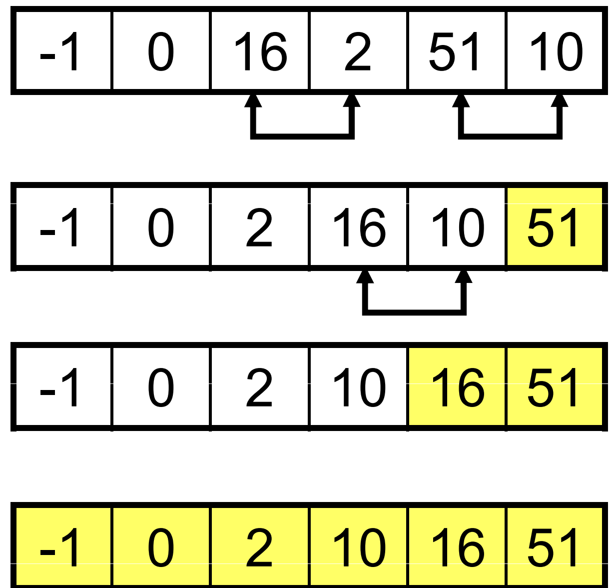
Può succedere che una *passata* dell’algoritmo ordini più di un elemento



• ad ogni *passata* l’ultimo elemento scambiato e tutti quelli alla sua destra sono sicuramente ordinati

Miglioramenti del bubble sort

Può succedere che in una *passata* dell'algoritmo non avvengano scambi



- se in una *passata* del bubble sort non avvengono scambi, allora l'array è ordinato

Implementazione del bubble sort

```
/* Ordinamento a bolle */
  public static void bubbleSort(int[] dati) {
        // pre: dati!=null
        // post: dati ordinato in modo non decrescente
        int n = dati.length;    // numero di elementi
        int i;                  // per la scansione di dati
        int ordinati;           // num. di elementi ordinati
        int ultimoScambio;      // posizione ultimo scambio
        ordinati = 0;
        while (ordinati<n) {
            ultimoScambio = 0;  // ipotizza nessuno scambio
            for (i=1; i<n-ordinati; i++)
                if (dati[i]<dati[i-1]) {
                    scambia(dati, i, i-1);
                    ultimoScambio = i;
                }
            // a destra dell'ultimo scambio sono ordinati
            ordinati = n-ultimoScambio;
        }
    }
```

Complessità del bubble sort

Qual è l' *operazione dominante* per il bubble sort?

- l'operazione dominante è il confronto tra coppie di elementi (dati[i]<dati[i-1])

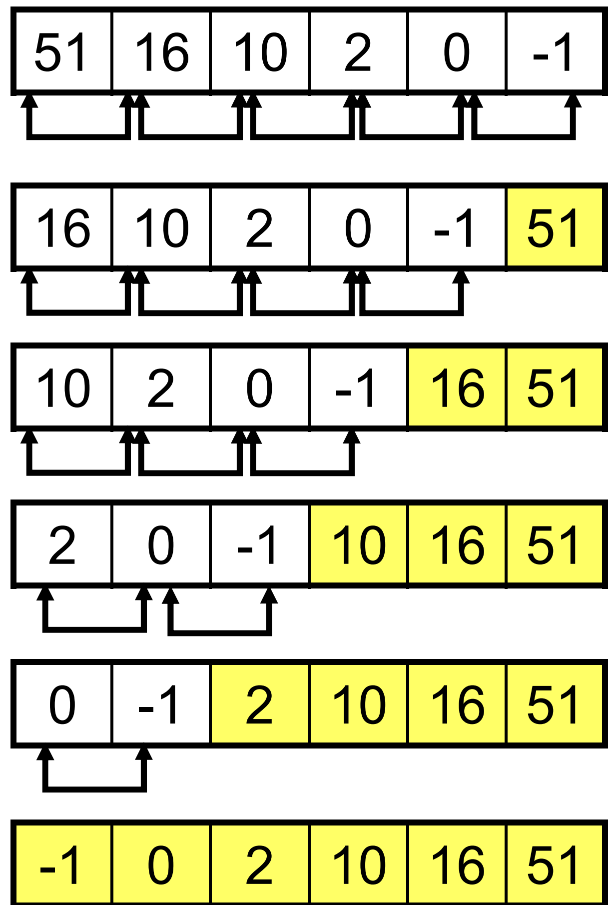
Qual è il *caso peggiore* per il bubble sort?

- il caso peggiore si verifica quando l'array è ordinato in modo non crescente

Quante volte viene eseguita l'operazione dominante nel caso peggiore?

Caso peggiore del bubble sort

Il caso peggiore si verifica quando l’array è ordinato in modo non crescente



Il bubble sort ha complessità quadratica

Esecuzione dell' operazione dominante nel caso peggiore:

- durante la prima passata vengono eseguiti N-1 confronti e altrettanti scambi
- durante la seconda passata vengono eseguiti N-2 confronti e altrettanti scambi
- .....
- durante la N-1-esima passata viene eseguito 1 confronto e 1 scambio

Il numero di confronti (e scambi) è $\sum_{i=1}^{N-1} (N-i) = N\,(N-1)/2$

Quindi, il bubble sort ha complessità asintotica $O(N^2)$ - cioè quadratica

Caso migliore del bubble sort

E' facile verificare che il *caso migliore* per il bubble sort si presenta quanto l'array è già ordinato

- in questo caso, durante la prima passata non avvengono scambi e si può quindi concludere che l'array è ordinato
- nel caso migliore, il bubble sort è *O(N)*

Ordinamento veloce

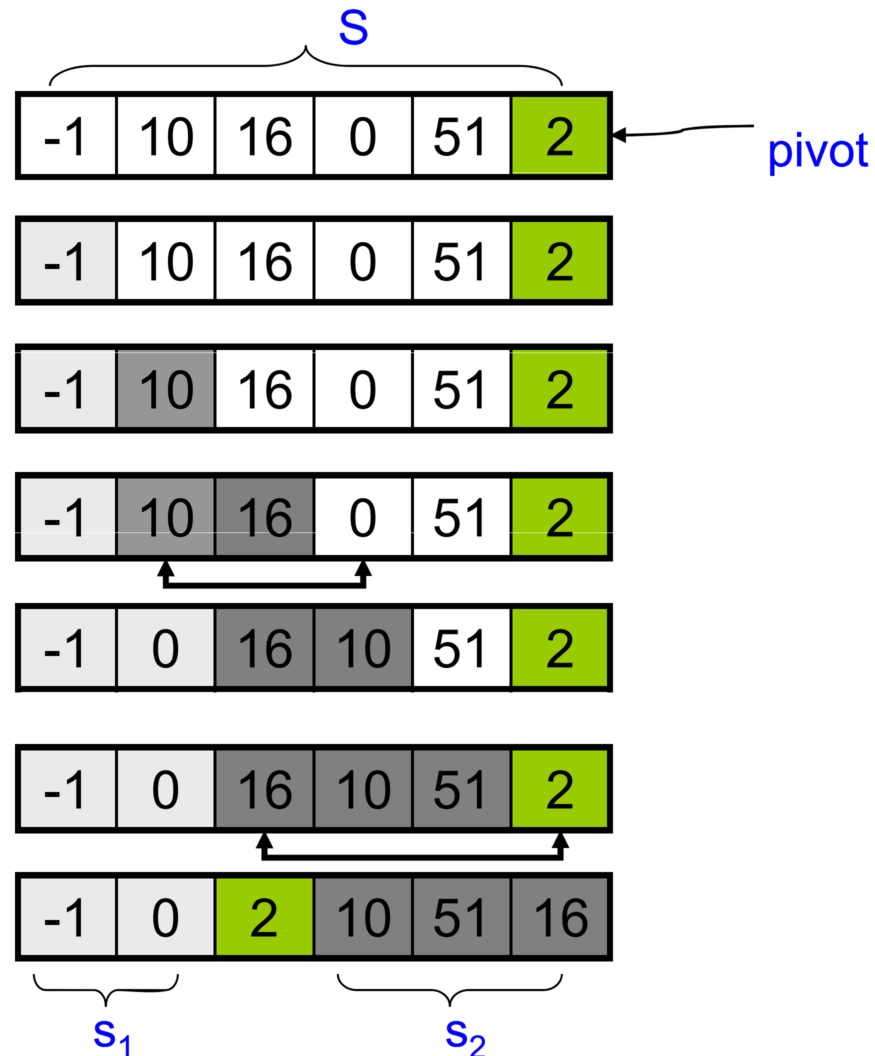
L'algoritmo di ordinamento veloce (quick sort) ordina una sequenza S di elementi sulla base della seguente strategia:

- viene scelto arbitrariamente un elemento della sequenza, chiamato pivot (ovvero perno)
- gli elementi della sequenza, ad eccezione del pivot, vengono suddivisi in due sottosequenze contigue: la sottosequenza s_1 degli elementi minori del pivot e la sottosequenza s_2 degli elementi maggiori del pivot
- le sottosequenze s_1 e s_2 vengono poi ordinate ricorsivamente
- la sequenza S ordinata è formata dalla sottosequenza s_1 ordinata, seguita dal pivot, seguita dalla sottosequenza s_2 ordinata

Comportamento del quick sort

Ecco come avviene la prima partizione dell'array:

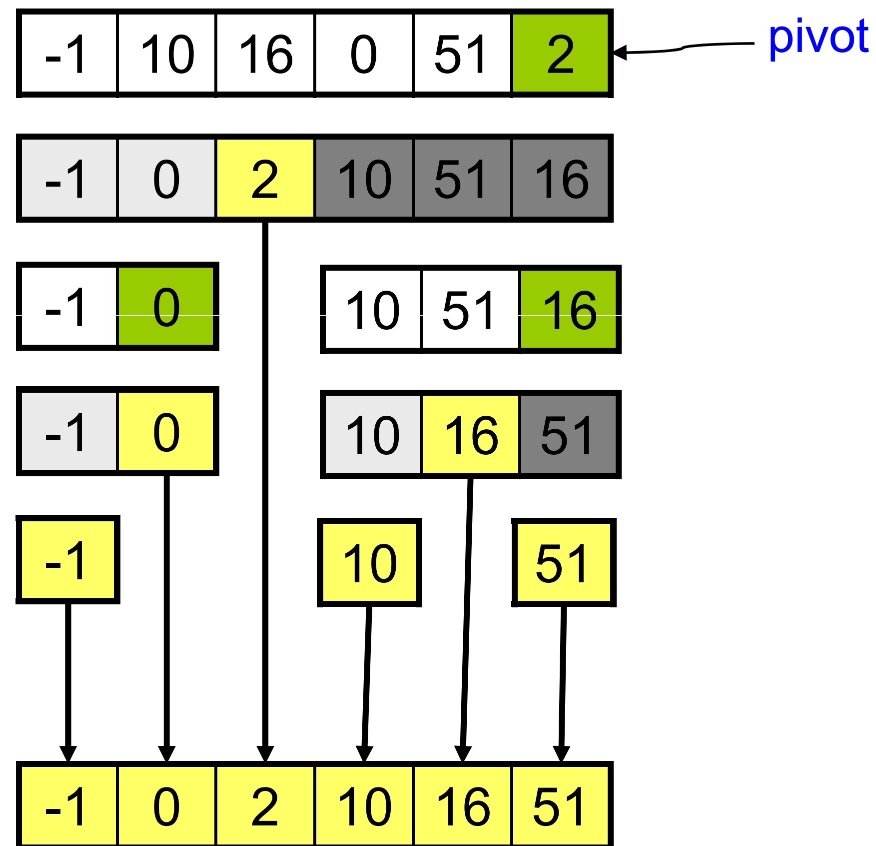
Gli elementi dell'array su sfondo grigio chiaro sono nella partizione s_1, quelli con sfondo grigio scuro sono nella partizione s_2, quelli con sfondo bianco devono essere valutati, l'ultimo elemento su sfondo verde è il pivot



Comportamento del quick sort

Ecco come viene ordinato l’intero array:

Il pivot, dopo la suddivisione di S nelle sottosequenze s_1 e s_2, viene collocato nella sua posizione definitiva (mostrata in giallo), con tutti gli elementi minori alla sua sinistra e i maggiori alla sua destra



Descrizione del quick sort

La strategia del quick sort per ordinare il generico sottoarray *A[p…r]* si basa dunque sulle seguenti fasi:

- partiziona l'array *A[p…r]* in due sottoarray *A[p…q-1]* e *A[q+1…r]* (eventualmente vuoti) tali che ogni elemento di *A[p…q-1]* sia minore o uguale ad *A[q]* che a sua volta è minore o uguale a ogni elemento *A[q+1…r]*
- ordina ricorsivamente i due sottoarray *A[p…q-1]* e *A[q+1…r]*

Implementazione del quick sort

Il quick sort può essere implementato utilizzando tre metodi:

- *quickSort*: metodo che avvia la ricorsione e alloca alcune risorse condivise
- *quickSortRic*: il metodo ricorsivo, per ordinare l'array dati
- *partition*: metodo che partiziona il sottoarray

Alcune considerazioni:

- ciascuna attivazione di *quickSortRic* ordina una porzione contigua di dati, delimitata da due indici che gli vengono passati come parametri
- il metodo *partition* partiziona un sottoarray e calcola e restituisce l'indice *q*, ovvero l'indice di *dati* in cui viene collocato definitivamente l'elemento *pivot* corrente

Implementazione del quick sort

Ecco il metodo che alloca le risorse e avvia la ricorsione, chiedendo a *quickSortRic* di ordinare l'intero array

```
/* Ordinamento veloce */
public static void quickSort(int[] dati) {
    int n = dati.length;   // numero di elementi

    /* avvia la ricorsione */
    quickSortRic(dati, 0, n-1);
        // 0 e n-1 delimitano la porzione
        // di array da ordinare (cioè tutto)
}
```

Implementazione del quick sort

Ecco il metodo ricorsivo in cui ciascuna attivazione è responsabile dell'ordinamento degli elementi di *dati* di indice compreso tra *p* e *r*

```
private static void quickSortRic(int[] dati,int p,int r){
    if(p<r){
            /* calcola l'indice in cui è posizionato
            definitivamente l'elemento pivot corrente*/
            int q = partition(dati,p,r);

            /* ordina ricorsivamente i due sottoarray */
            quickSortRic(dati,p,q-1);
            quickSortRic(dati,q+1,r);
            }
}
```

Implementazione del quick sort

Ecco il metodo che partiziona gli elementi di *dati* di indice compreso tra *p* e *r*

```
private static int partition(int[]dati,int p,int r){
    int pivot = dati[r];
    int i = p-1; // posizione dell'ultimo elemento di s1
    int j;
    for(j=p; j<=r-1; j++)
          if(dati[j]<=pivot){
                i = i + 1;
                scambia(dati,i,j);
          }
    /* colloca l'elemento pivot nella sua posizione
    definitiva che ha indice i+1 */
    scambia(dati,i+1,r);
    return i+1;
}
```

Complessità del quick sort

Si può dimostrare che nel caso peggiore il quick sort ha complessità asintotica $O(N^2)$ - cioè quadratica

Tuttavia il quick sort, ha nel “caso medio”, complessità $O(NlogN)$ ed è sperimentalmente molto veloce

Ordinamento per fusione

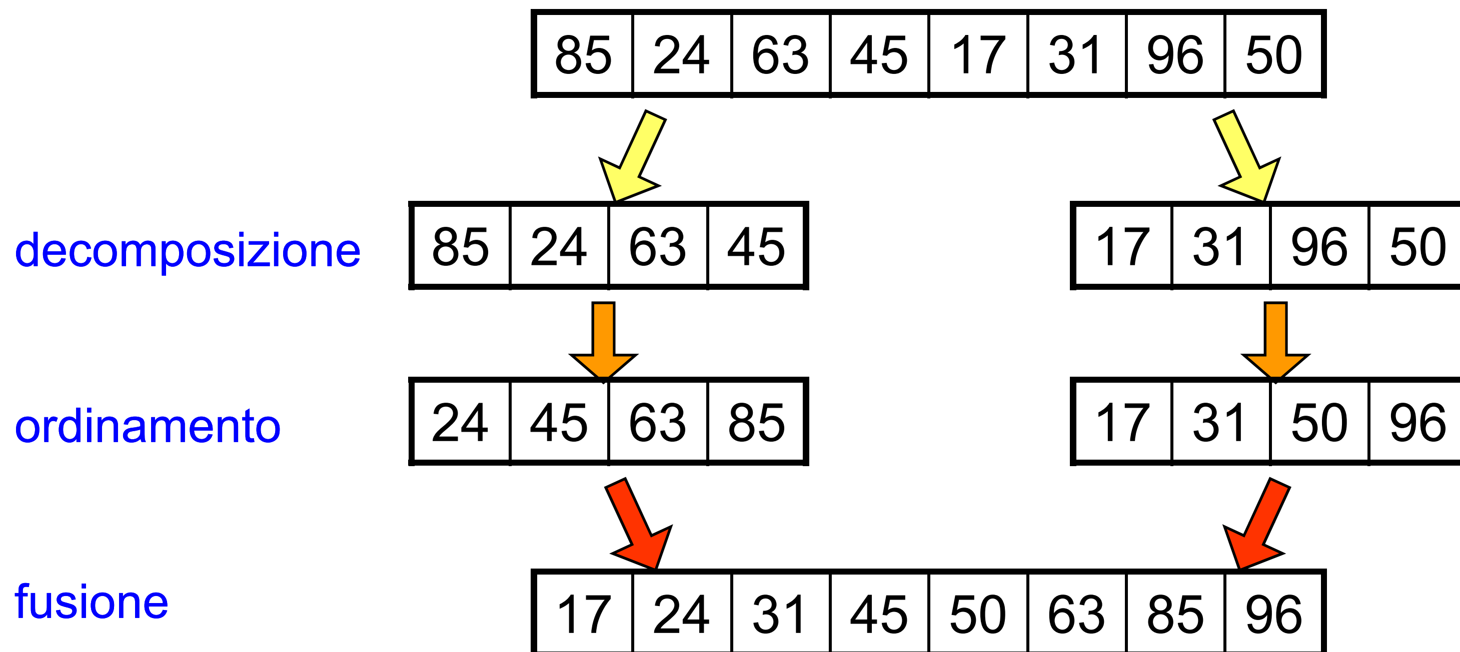
L'algoritmo di ordinamento per fusione (merge sort) implementa la seguente strategia ricorsiva:

- se l'array da ordinare contiene uno o due elementi, allora l'array viene ordinato direttamente
  - mediante al più un confronto e uno scambio
- se l'array da ordinare contiene più di due elementi, allora viene ordinato come segue
  - gli elementi dell'array vengono partizionati in due sequenze
  - ciascuna sequenza viene ordinata separatamente
  - le due sequenze ordinate vengono "fuse" in un'unica sequenza ordinata

Strategia del merge sort

La strategia del merge sort si basa su tre fasi:

- decomposizione dell’array in due sequenze
- ordinamento delle due sequenze
- fusione delle due sequenze in un’unica sequenza ordinata



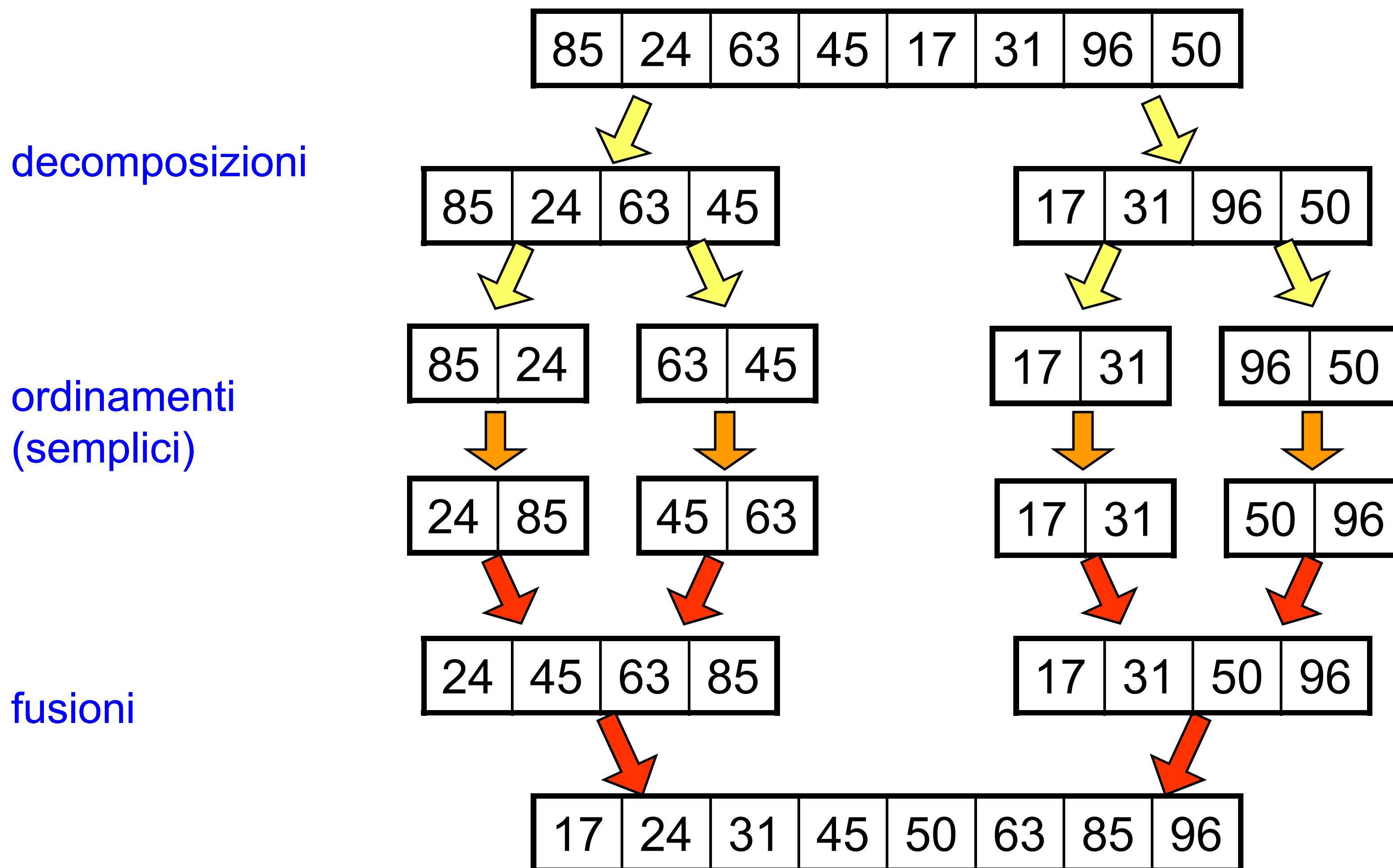
Il merge sort è un algoritmo ricorsivo

L'implementazione del merge sort richiede quindi di:

- stabilire una strategia di decomposizione dell'array
  - viene scelto di decomporre gli elementi dell'array in due sottosequenze contigue (in tal modo è possibile decomporre ricorsivamente le sequenze contigue in sottosequenze contigue)
- implementare un algoritmo di ordinamento
  - la scelta ovvia è quella di ordinare le due sottosequenze utilizzando ancora il merge sort (il merge sort è un algoritmo ricorsivo)
- implementare un algoritmo di fusione

Il merge sort – un esempio

Ecco una illustrazione più dettagliata della strategia del merge sort:



Implementazione del merge sort

Il merge sort può essere implementato utilizzando tre metodi:

- *mergeSort*: metodo che avvia la ricorsione e alloca alcune risorse condivise
- *mergeSortRic*: il metodo ricorsivo, per ordinare l'array dati
- *merge*: metodo che fonde due sottosequenze ordinate

Alcune considerazioni:

- ciascuna attivazione di *mergeSortRic* ordina una porzione contigua di dati, delimitata da due indici che gli vengono passati come parametri
- il metodo *merge* opera su sottosequenze di dati ordinate e contigue, e usa un array di appoggio *temp* condiviso da tutte le attivazioni
  - l'array *temp* ha la stessa lunghezza di *dati* e viene allocato da *mergeSort*

Il metodo mergeSort

Ecco il metodo che alloca le risorse e avvia la ricorsione, chiedendo a *mergeSortRic* di ordinare l’intero array

```
/* Ordinamento per fusione */
public static void mergeSort(int[] dati) {
    int n = dati.length;    // numero di elementi
    int[] temp;
    /* crea l'array di appoggio condiviso */
    temp = new int[n];      // stessa lunghezza di dati
    /* avvia la ricorsione */
    mergeSortRic(dati, temp, 0, n-1);
        // 0 e n-1 delimitano la porzione
        // di array da ordinare (cioè tutto)
}
```

Il metodo mergeSortRic

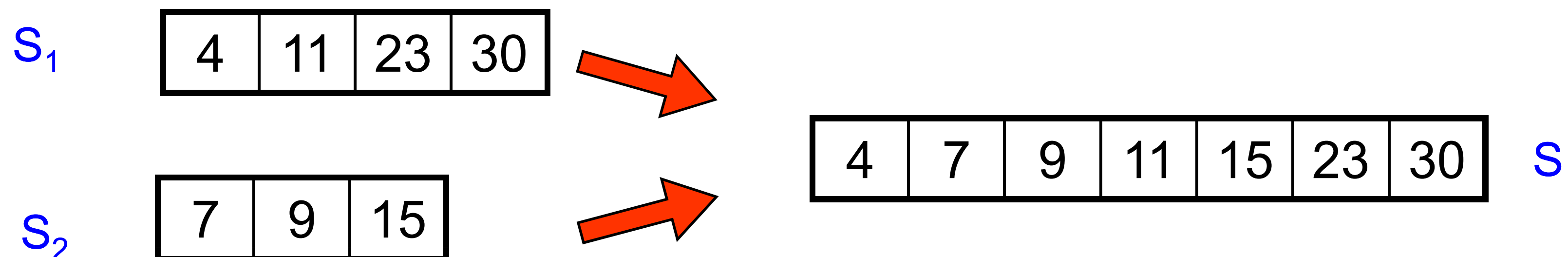
Ecco il metodo ricorsivo, in cui ciascuna attivazione ordina gli elementi di *dati* di indice compreso tra *sinistra* e *destra* (inclusi)

```
private static void mergeSortRic(int[] dati, int[] temp,
                                 int sinistra, int destra) {
        int n;          // elementi da ordinare
        int centro;     // per una ulteriore suddivisione
        n = destra-sinistra+1;
        if (n==2) { // caso base, confronto e scambio
            if (dati[sinistra]>dati[destra])
                scambia(dati, sinistra, destra);
        }
        else if (n>2) {// caso ricorsivo
            centro = (sinistra+destra)/2;
            mergeSortRic(dati, temp, sinistra, centro);
            mergeSortRic(dati, temp, centro+1, destra);
            merge(dati, temp, sinistra, centro, destra);
        }
        // se n<2, la sottosequenza è già ordinata
}
```

Fusione di sequenze ordinate

Per completare l'implementazione dell'ordinamento per fusione è necessario risolvere il problema della fusione di sequenze ordinate definito come segue:

- date due sequenze S_1 e S_2 ordinate in modo non decrescente, costruire una nuova sequenza S che contiene tutti gli elementi di S_1 e S_2 ed è ordinata in modo non decrescente



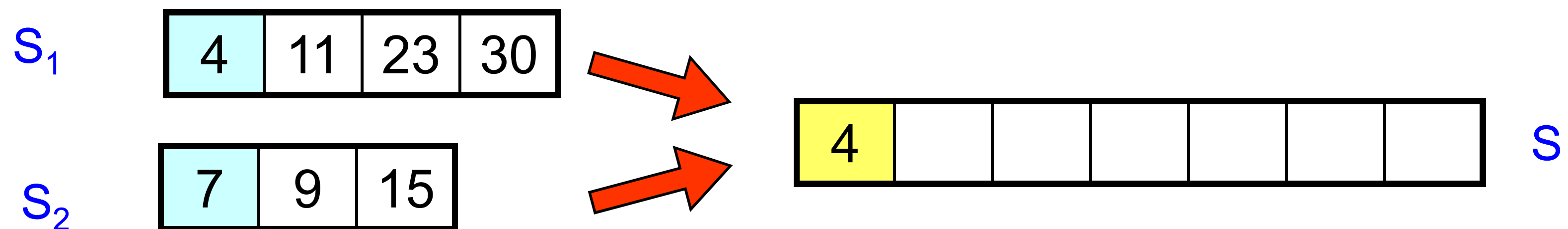
- le sequenze S_1 e S_2 possono avere lunghezza diversa
- la lunghezza di S è la somma delle lunghezze di S_1 e S_2

Fusione di sequenze ordinate

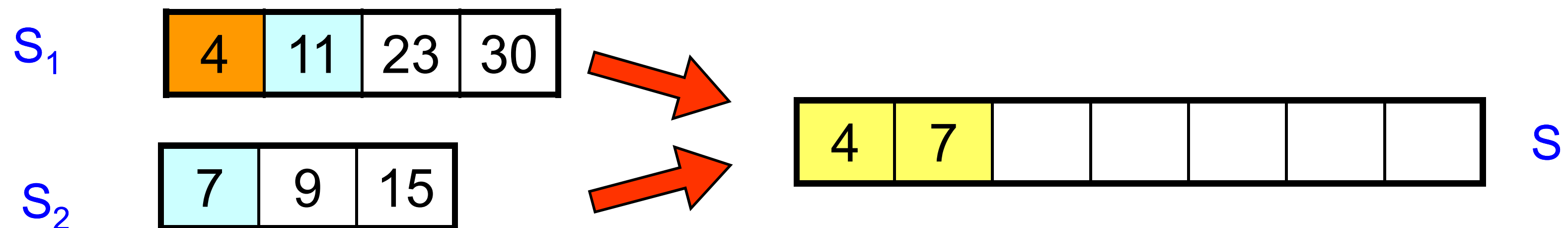
Il problema della fusione è risolto dal seguente algoritmo:

- seleziona gli elementi di S, uno alla volta in sequenza
- il prossimo elemento di S viene scelto tra l'elemento più piccolo di S_1 e quello più piccolo di S_2 tra quelli che non sono ancora stati inseriti in S

– selezione del primo elemento di S



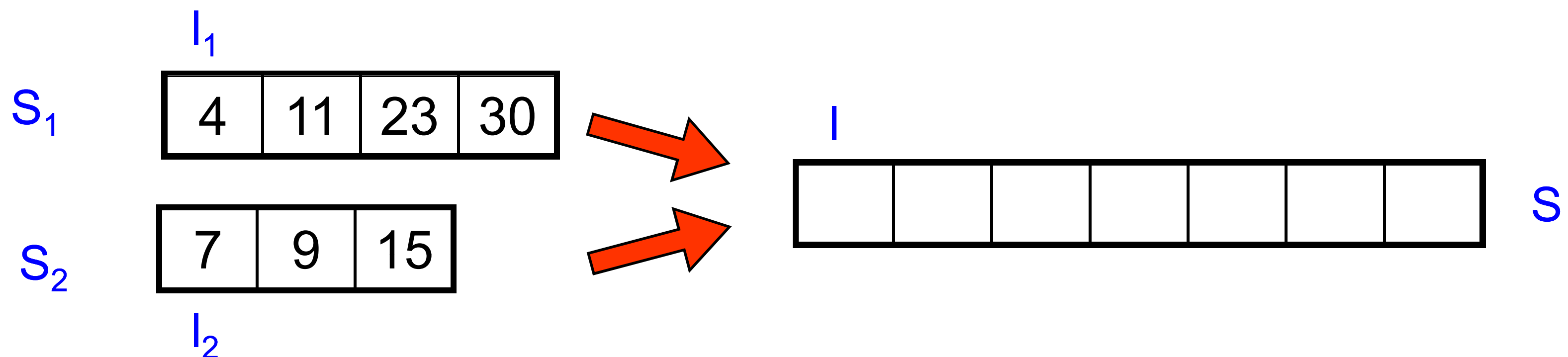
– selezione del secondo elemento di S



Fusione di sequenze ordinate

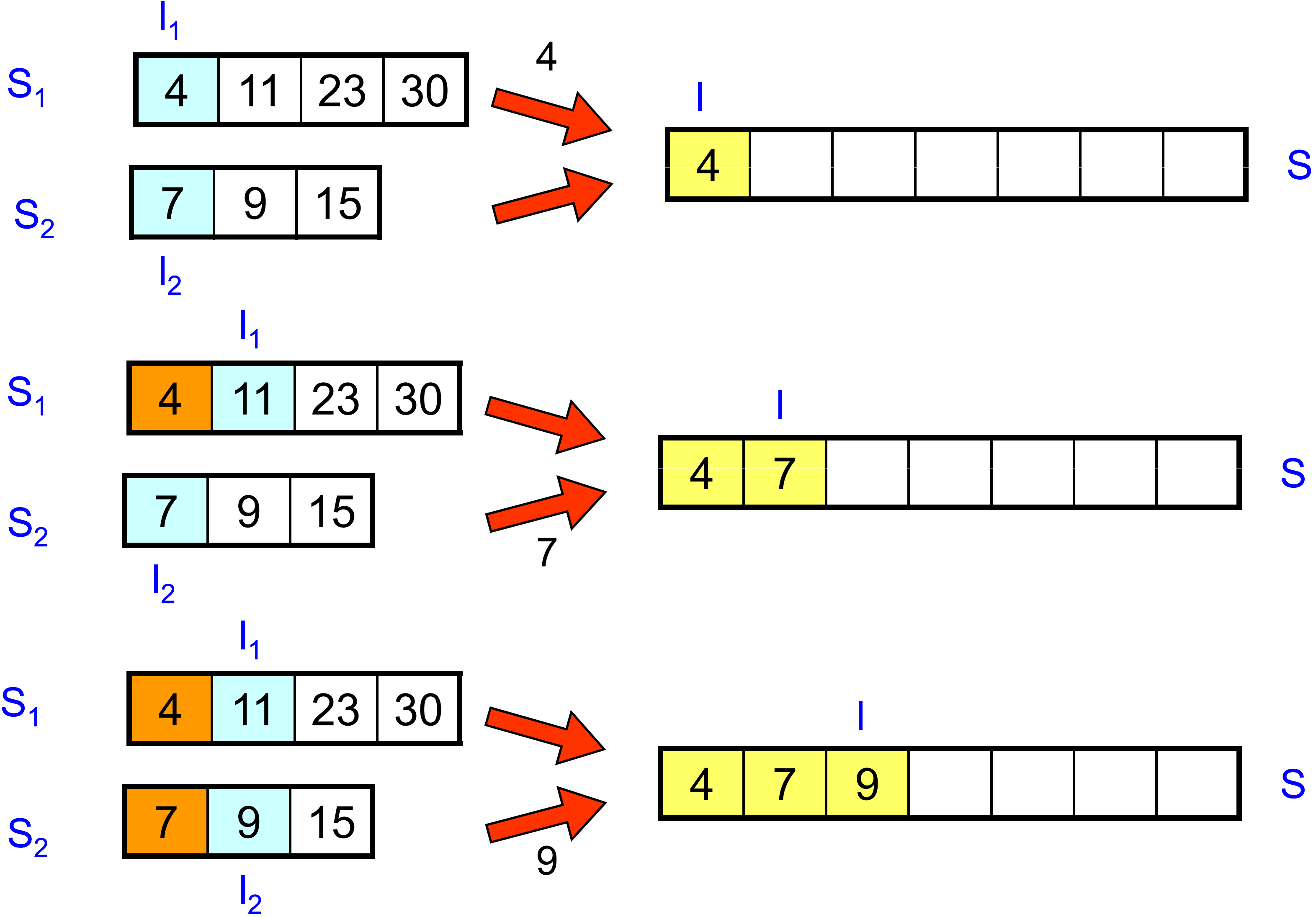
L'algoritmo per la fusione di array ordinati gestisce tre indici:

- l'indice I_1 del più piccolo elemento di S_1 tra quelli che non sono ancora stati inseriti in *S*
- l'indice I_2 del più piccolo elemento di S_2 tra quelli che non sono ancora stati inseriti in *S*
- l'indice *I* del prossimo elemento da inserire in *S*
    - I tre indici valgono inizialmente 0

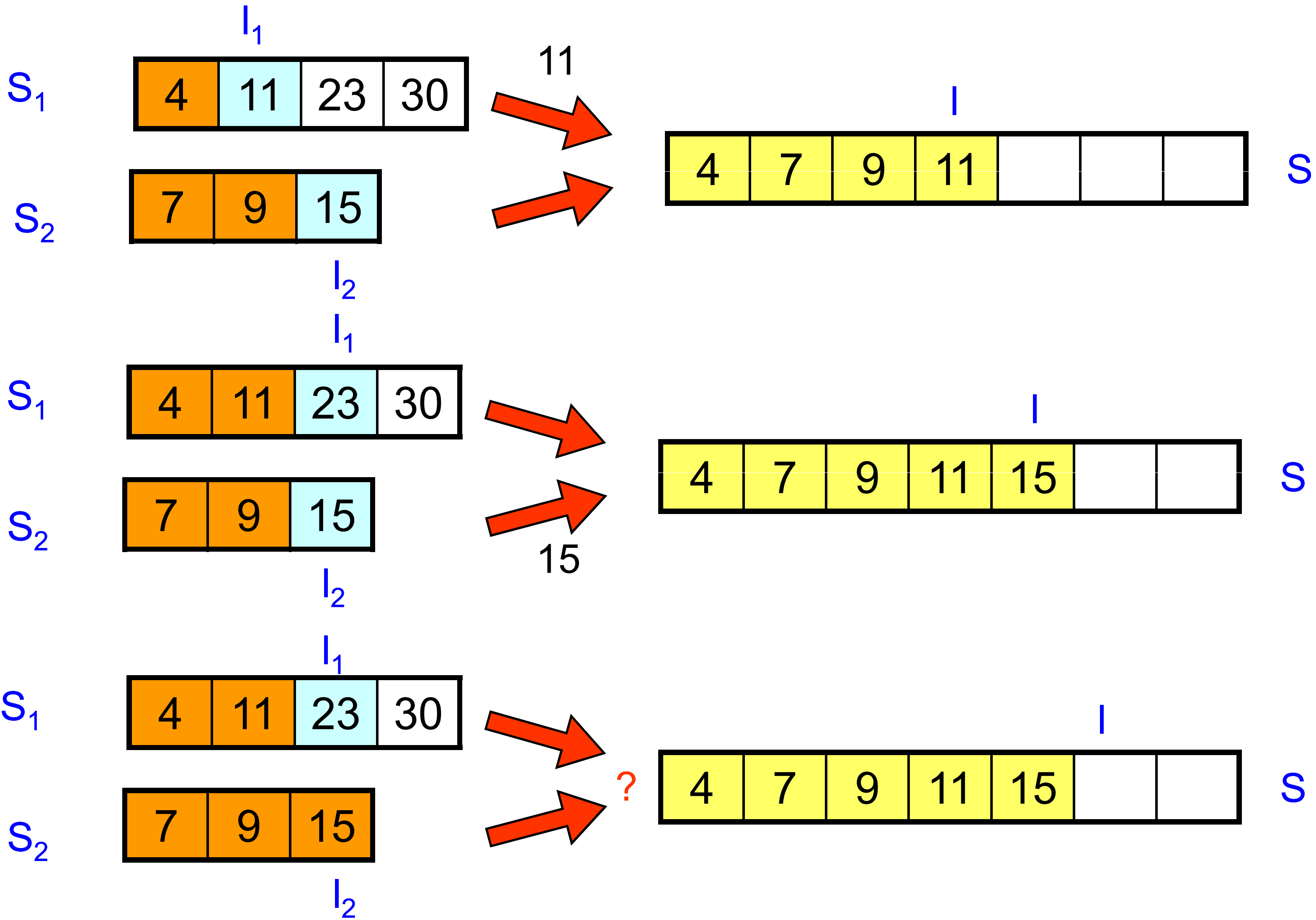


Fusione di sequenze ordinate - esempio

Ecco come avviene la fusione di due array:

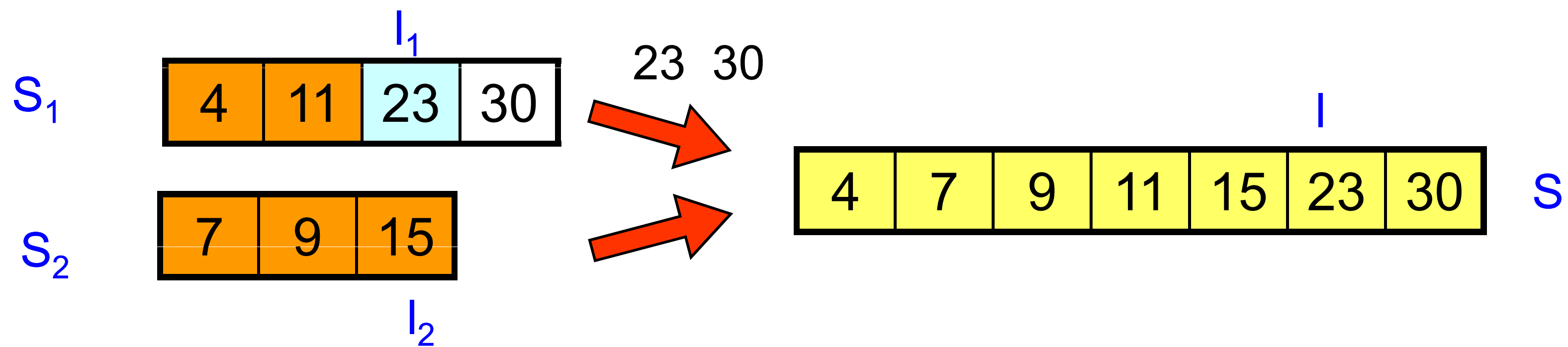


Fusione di sequenze ordinate - esempio



Fusione di sequenze ordinate - esempio

Quando tutti gli elementi di una sequenza sono stati inseriti in *S*, allora tutti i rimanenti elementi dell'altra sequenza possono essere inseriti in *S* senza ulteriori confronti:



Implementazione della fusione

Il seguente metodo implementa l'algoritmo di fusione per due array di interi ordinati in modo non decrescente

```
/* Fonde gli array s1 e s2 ordinati in modo
* non decrescente in un nuovo array ordinato. */

public static int[] fusione(int[] s1, int[] s2) {
      // pre: s1!=null && s2!=null &&
      //      s1 e s2 ordinati in modo non decrescente
      int[] s;                // risultato della fusione
      int i1, i2, i;          // indici per s1, s2, s
      /* inizializzazioni */
      s = new int[s1.length + s2.length];
      i1 = 0;
      i2 = 0;
      i = 0;
```

... segue ...

Implementazione della fusione

In una prima fase bisogna scegliere l'array da cui estrarre l'elemento da inserire in *S*

... segue ...

```
/* s1 e s2 contengono elementi ancora non in s */
while (i1<s1.length && i2<s2.length) {
    if (s1[i1]<s2[i2]) {
        s[i] = s1[i1];
        i1++;
    } else {  // allora s1[i1]>=s2[i2]
        s[i] = s2[i2];
        i2++;
    }
    i++;
}
// i1==s1.length || i2==s2.length
```

... segue ...

Implementazione della fusione

Poi bisogna inserire in *S* gli elementi residui dell'array che non è stato scandito interamente

... segue ...

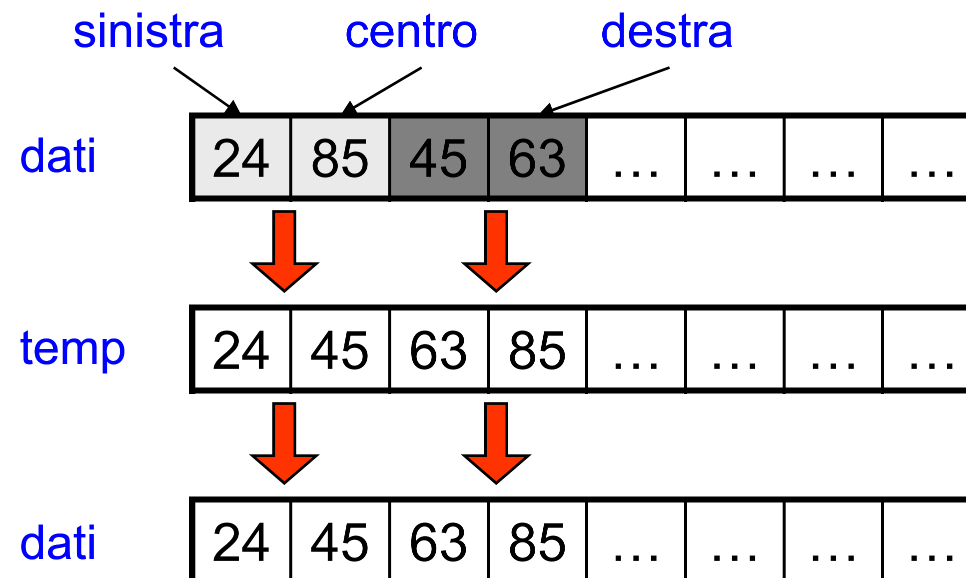
```
    /* il corpo viene eseguito se s1 non era
     * stato scandito interamente */
    while (i1<s1.length) {
        s[i] = s1[i1];
        i1++; i++;
    }
    /* il corpo viene eseguito se s2 non era
     * stato scandito interamente */
    while (i2<s2.length) {
        s[i] = s2[i2];
        i2++; i++;
    }
    // tutto è stato fatto
    return s;
}
```

Fusione nel merge sort

Nel caso del *merge sort*:

- le due sequenze ordinate sono sottosequenze contigue dell'array *dati*, delimitate da indici passati come parametri

- la sequenza ordinata risultato viene temporaneamente memorizzata nell'array di appoggio *temp* e poi copiata in *dati*



Il metodo merge

Ecco il metodo *merge* che implementa l'algoritmo di fusione per il *merge sort*

```
/* fusione di sottosequenze contigue ordinate */
  private static void merge(int[] dati, int[] temp,
          int sinistra, int centro, int destra) {
      // pre: le seguenti sottosequenze di dati
      //      sono ordinate in modo non decrescente:
      //      da sinistra a centro
      //      da centro+1 a destra
      // post: dati ordinato da sinistra a destra
      int i1, i2, i;       // indici per le sottosequenze
      /* inizializzazioni */
      i1 = sinistra;
      i2 = centro+1;
      i = sinistra;
```

... segue ...

Il metodo merge

Prima fase

... segue ...

```
/* entrambe le sequenze hanno elementi residui */
while (i1<=centro && i2<=destra) {
    if (dati[i1]<dati[i2]) {
        temp[i] = dati[i1];
        i1++;
    } else {  // allora dati[i1]>=dati[i2]
        temp[i] = dati[i2];
        i2++;
    }
    i++;
}
// i1>centro || i2>destra
```

... segue ...

Il metodo merge

Scansione degli elementi residui e ricostruzione di *dati*

... segue ...

```
/* prima sottosequenza scandita non interamente */
while (i1<=centro) {
        temp[i] = dati[i1];
        i1++; i++;
    }
/* seconda sottosequenza scandita non interamente */
while (i2<=destra) {
        temp[i] = dati[i2];
        i2++; i++;
    }

/* ora copia da temp a dati */
for (i=sinistra; i<=destra; i++)
        dati[i] = temp[i];
    // tutto fatto !
}
```

Complessità del merge sort

Qual è la *complessità asintotica* del merge sort?

- non abbiamo gli strumenti metodologici per studiare la complessità di algoritmi ricorsivi
- non è semplice riscrivere l'algoritmo di merge sort in modo iterativo
- ci baseremo su considerazioni di massima — ma corrette

Qual è l'*operazione dominante* del merge sort?

- l'operazione dominante del merge sort è la fusione di sottosequenze ordinate
- quante volte viene eseguita questa operazione?
- quanto costa ciascuna esecuzione di questa operazione?

Complessità del merge sort

Qual è la complessità della fusione di sequenze ordinate?

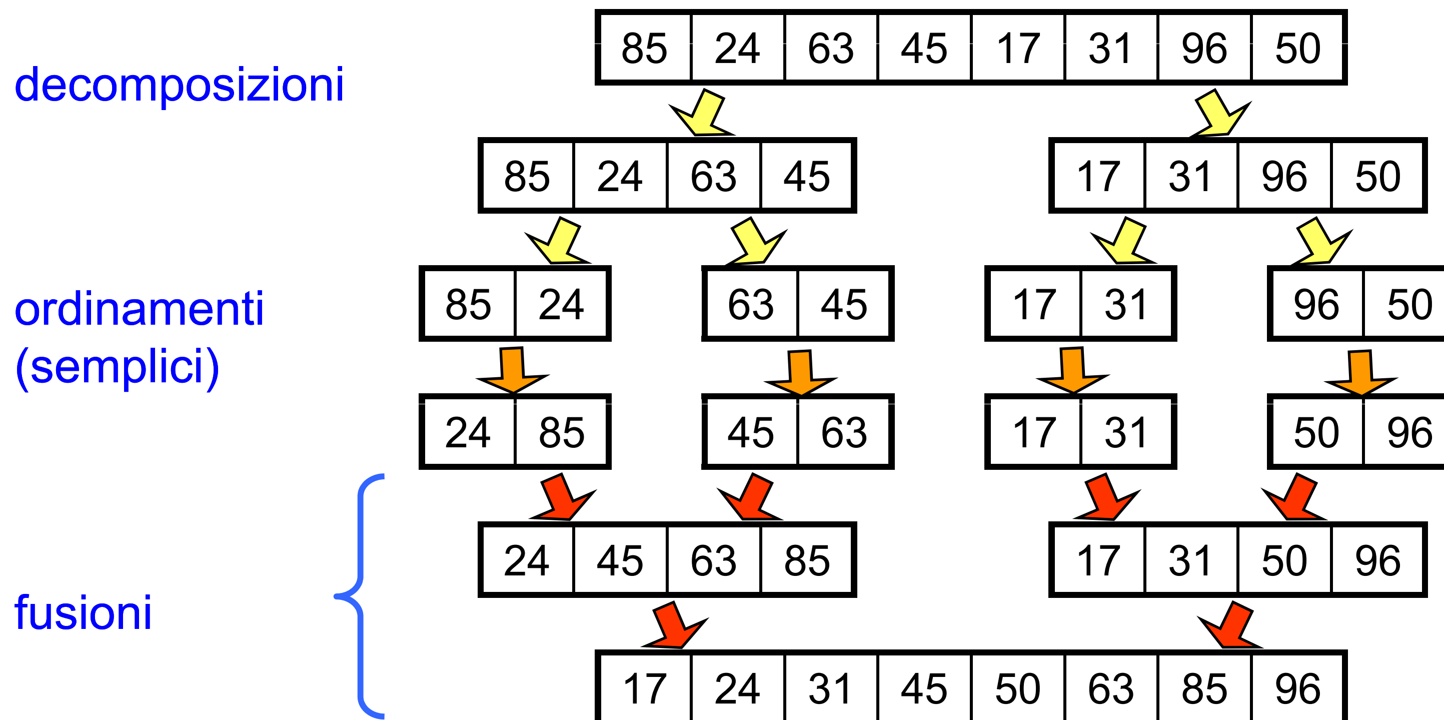
- l'*operazione dominante* (in ciascuna fase) è l'assegnazione di un elemento del risultato
- la complessità asintotica della fusione è data dal numero di elementi della sequenza calcolata

Quante volte viene eseguita la fusione nel merge sort?

Complessità del merge sort

La figura mostra che nel merge sort l'array viene decomposto e fuso interamente un certo numero di volte

- nell'esempio possiamo identificare tre livelli di decomposizioni e tre livelli di fusioni



Il merge sort ha complessità logaritmica

Entro ciascun livello, il costo delle fusioni è *lineare* nella dimensione dell'intero array

Il numero di livelli di decomposizioni e fusioni è logaritmico nella dimensione dell'array

- il numero di livelli di decomposizioni e fusioni è $log_2 N$

La complessità asintotica del merge sort è quindi $O(N \log N)$

- il merge sort è asintoticamente più efficiente degli altri algoritmi di ordinamento studiati

Qual'è il miglior algoritmo di ordinamento?

E' possibile dimostrare che non è possibile definire algoritmi di ordinamento basati su confronti e scambi che hanno complessità asintotica migliore di *O(N log N)*

- il *merge sort* è un algoritmo di ordinamento ottimale

In pratica, l'algoritmo di ordinamento di array più usato è il *quick sort*

- il quick sort ha complessità asintotica $O(N^2)$

Perché il *quick sort* è preferito al *merge sort* ?

- il comportamento "medio" del quick sort è *O(NlogN)*
- nel caso "medio", il costo esatto del quick sort ha un fattore moltiplicativo minore del merge sort

Alcuni risultati sperimentali

La seguente tabella mostra alcuni dati sperimentali (tempi medi in secondi) di alcuni metodi di ordinamento

N Algoritmo	1000	10000	100000	1000000
selection sort	0.02	0.27	24.05	circa 40 minuti
insertion sort	0.01	0.23	18.80	circa 30 minuti
bubble sort (semplice)	0.01	0.66	62.35	circa 100 minuti
bubble sort	0.02	0.67	62.20	circa 100 minuti
quick sort	0.01	0.02	0.05	0.35
merge sort	0.01	0.02	0.05	0.51

sperimentalmente si verifica che il *quick sort* è mediamente più veloce del *merge sort*

Glossario dei termini principali

Termine	Significato
Array ordinato (in modo non decrescente)	Array ordinato in modo tale che ogni suo elemento è maggiore o uguale agli elementi che lo precedono
Strategia di ordinamento per confronti e scambi	Strategia di ordinamento basata sul confronto fra coppie di elementi, seguito da scambi di elementi che non sono ordinati fra loro. I vari algoritmi di ordinamento si differenziano su come vengono selezionate (e scambiate) le coppie di elementi da confrontare