Programmazione 2
e Laboratorio di Programmazione

Corso di Laurea in

Informatica

Università degli Studi di Napoli "Parthenope"

Anno Accademico 2023-2024

Prof. Luigi Catuogno

1

Informazioni sul corso

Docente	Luigi Catuogno **luigi.catuogno@uniparthenope.it**
Orario	Lun: 9:00-11:00 Mer: 11:00-13:00
Sede	Centro Direzionale Napoli **Aula Magna**
Ricevimento	Mer: 14:00-16:00 (previo appuntamento) Ufficio docente oppure Team: **cxxa3bo**

2

Libri di testo

Introduzione al linguaggio – costrutti e tecniche di base

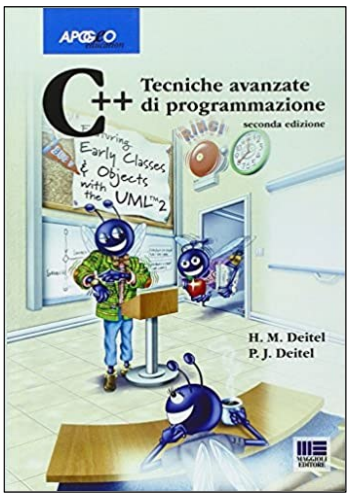
[FdP] H. M. Deitel, P. J. Deitel
C++ Fondamenti di programmazione
II ed. (2014) Maggioli Editore (Apogeo Education)
ISBN: 978-88-387-8571-9

3

Libri di testo

Tecniche avanzate e strutture dati elementari

[TAP] H. M. Deitel, P. J. Deitel
C++ Tecniche avanzate di programmazione
II ed. (2011) Maggioli Editore (Apogeo Education)
ISBN: 978-88-387-8572-6



4

Risorse on-line

Team del corso

Programmazione 2 AA 2023-24 - Prof. Catuogno
Comunicazioni, incontri e avvisi per il corso
Codice: **ftomzjx**

Piattaforma e-learning

Programmazione II e Laboratorio di Programmazione II - A.A. 2023-24
Materiale didattico, manualistica, esercitazioni.
URL: https://elearning.uniparthenope.it/course/view.php?id=2386

5

Algoritmi di ordinamento

6

Il problema dell'ordinamento

- L'ordinamento degli elementi di un insieme è un problema classico dell'informatica
- Rientra in una innumerevole quantità di applicazioni ed è spesso un passo imprescindibile nella soluzione di problemi ben più complicati
- E' anche un utile strumento didattico: il problema in se è molto semplice e chiunque è in grado di comprenderne i termini essenziali
- Per l'ordinamento sono stati proposti numerosi algoritmi molto eleganti che consentono di evidenziare gli aspetti fondamentali della progettazione e della costruzione di un algoritmo efficiente

7

Il problema dell'ordinamento

- Dato un insieme di n numeri $\{a_1, a_2, \dots, a_n\}$, trovare un'opportuna permutazione $\{a'_1, a'_2, \dots, a'_n\}$ tali che: $a'_1, \leq a'_2, \leq \cdots \leq a'_n$

Input: $\{a_1, a_2, \dots, a_n\}$

Output: $\{a'_1, a'_2, \dots, a'_n\}$

oppure

$\{a_{\pi(1)}, a_{\pi(2)}, \dots a_{\pi(n)}\}$

dove π è una opportuna permutazione degli indici *1,…, n*

8

SelectionSort

9

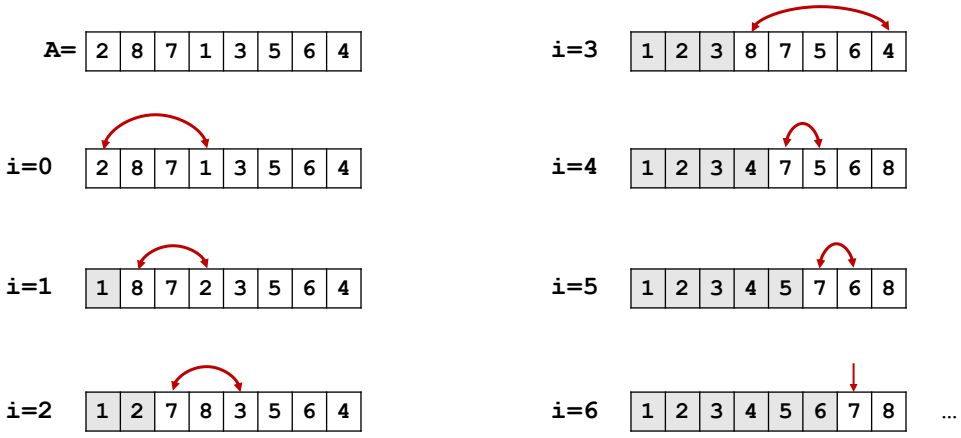
SelectionSort

Ricordiamo brevemente l'algoritmo di ordinamento per *selezione* (*selectionSort*). Sia **A[]** un array di ***n*** interi, il seguente algoritmo lo ordina in maniera crescente:

```
SelectionSort(A) {
        for(i=0;i<n<<;i++) {
                min=i;
                for (j=i+1;j<n;j++)
                        if (A[j]<A[min])
                                swap(A[min],A[i]);
        }
}
```

10

SelectionSort



11

Analisi di **SelectionSort**

- Dal punto di vista delle operazioni svolte, non esiste un caso particolarmente favorevole o, al contrario, particolarmente sfavorevole: l'algoritmo esegue lo stesso numero di operazioni qualunque sia la configurazione iniziale dell'array **A**;

  Ad ogni iterazione del ciclo più esterno (**i**), il ciclo più interno (**j**) esegue esattamente $(n - i)$ confronti. Il numero totale di confronti è :

$$\sum_{i=1}^{n-1}(n-i) = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} = \mathrm{O}(\mathrm{n}^2)$$

12

MergeSort

13

MergeSort

- **`MergeSort`** è un algoritmo di tipo *«divide-et-impera»**

  * Gli algoritmi di questa classe, sono concepiti per risolvere un problema dividendolo in sottoproblemi più semplici/ridotti, risolvendo questi ultimi e combinandone i risultati per ottenere la soluzione al problema di partenza.

    Si identificano le tre fasi: *Divide*, *Impera* e *Combina*.

    Si tratta di una tecnica molto utilizzata, *quasi* sempre nell'ambito di algoritmi ricorsivi.

14

MergeSort

- **MergeSort** è un algoritmo di tipo *«divide-et-impera».* Le tre fasi possono essere così descritte:
  - *Divide*: gli n elementi della sequenza da ordinare vengono in due sottosequenze (approssimativamente) di n/2 elementi ciascuna
  - *Impera:* ordina, usando ricorsivamente il **MergeSort**, le due sottosequenze
  - *Combina*: fonde le due sottosequenze per produrre come risposta la sequenza ordinata

15

MergeSort

Sia **A[]** un array di *n* interi, **sx** e **dx**, sono gli indici del primo e l'ultimo elemento della sottosequenza degli elementi di **A** da ordinare. All'inizio, **sx**=*0* e **dx**=*n-1:*

```
MergeSort(A,sx,dx) {
        if(sx<dx) {
                med=⌊(sx + dx)/2⌋;
                MergeSort(A,sx,med);
                MergeSort(A,med+1,dx);
                Merge(A,sx,med,dx);
        }
}
```

16

MergeSort

Sia **A[]** un array di *n* interi, **sx** e **dx**, sono gli indici del primo e l'ultimo elemento della sottosequenza degli elementi di **A** da ordinare. All'inizio, **sx**=*0* e **dx**=*n-1:*

```
MergeSort(A,sx,dx) {
        if(sx<dx) {
                med=⌊(sx + dx)/2⌋;
                MergeSort(A,sx,med);
                MergeSort(A,med+1,dx);
                Merge(A,sx,med,dx);
        }
}
```

Il processo di suddivisione si ferma quando la sequenza da ordinare ha lunghezza 1

17

MergeSort

Sia **A[]** un array di *n* interi, **sx** e **dx**, sono gli indici del primo e l'ultimo elemento della sottosequenza degli elementi di **A** da ordinare. All'inizio, **sx**=*0* e **dx**=*n-1:*

```
MergeSort(A,sx,dx) {
        if(sx<dx) {
                med=⌊(sx + dx)/2⌋;
                MergeSort(A,sx,med);
                MergeSort(A,med+1,dx);
                Merge(A,sx,med,dx);
        }
}
```

La fase di ricombinazione si avvale della funzione ausiliaria **Merge** per fondere le due sottosequenze…

18

MergeSort

Sia **A[]** un array di *n* interi, **sx** e **dx**, sono gli indici del primo e l'ultimo elemento della sottosequenza degli elementi di **A** da ordinare. All'inizio, **sx**=*0* e **dx**=*n-1:*

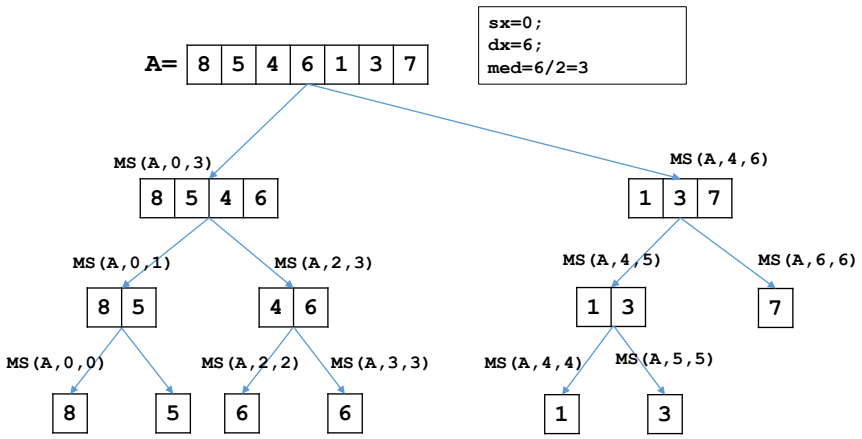
```
MergeSort(A,sx,dx) {
        if(sx<dx) {
                med=⌊(sx + dx)/2⌋;
                MergeSort(A,sx,med);
                MergeSort(A,med+1,dx);
                Merge(A,sx,med,dx);
        }
}
```

La fase di ricombinazione si avvale della funzione ausiliaria **Merge** per fondere le due sottosequenze…

La **Merge** procede assumendo che le due sottosequenze **A[sx..med]** e **A[med+1..dx]** siano già ordinate.

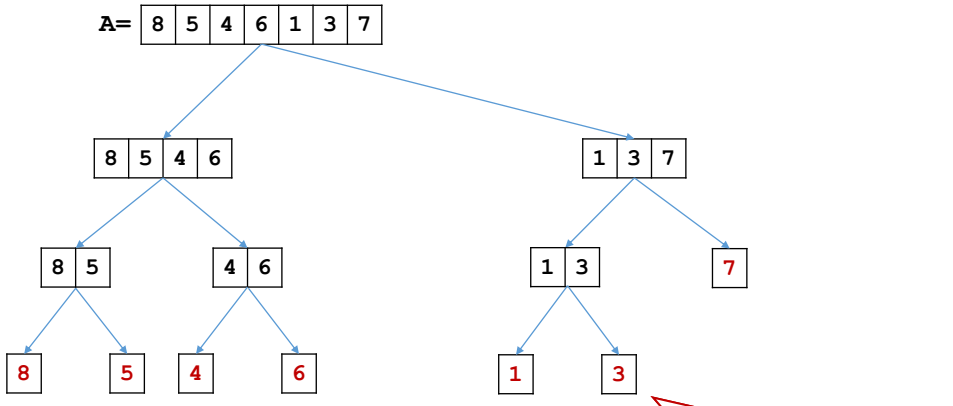
19

MergeSort



20

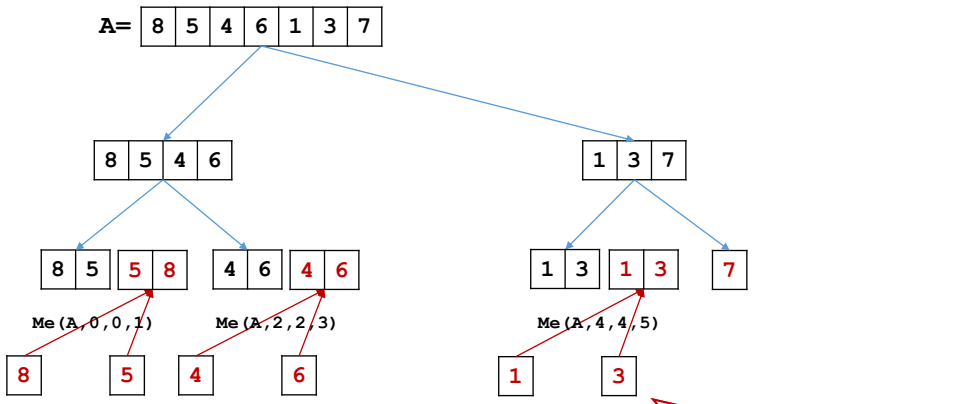
MergeSort



La sequenze di lunghezza 1 sono la *base della ricorsione* e si intendono già ordinate. La **Merge**, inizia a ricombinare questi «singoli» in sequenze da due…

21

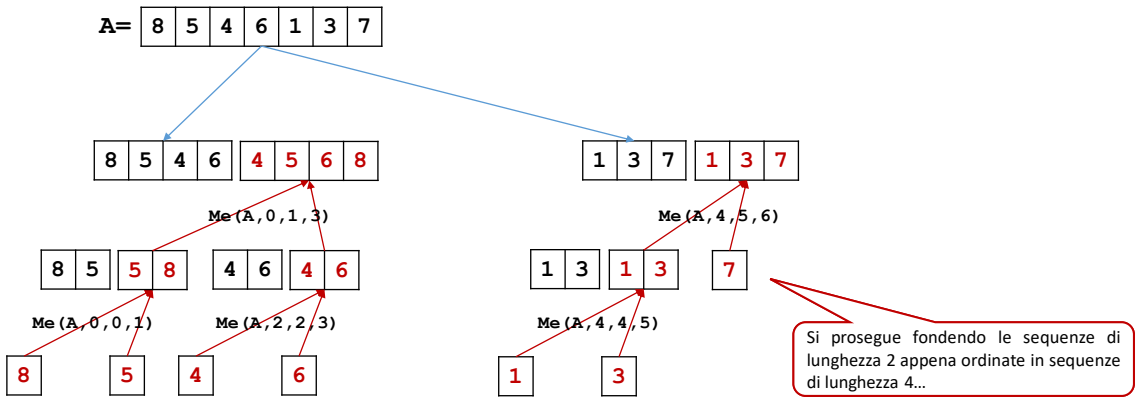
MergeSort



La sequenze di lunghezza 1 sono la *base della ricorsione* e si intendono già ordinate. La **Merge**, inizia a ricombinare questi «singoli» in sequenze da due…

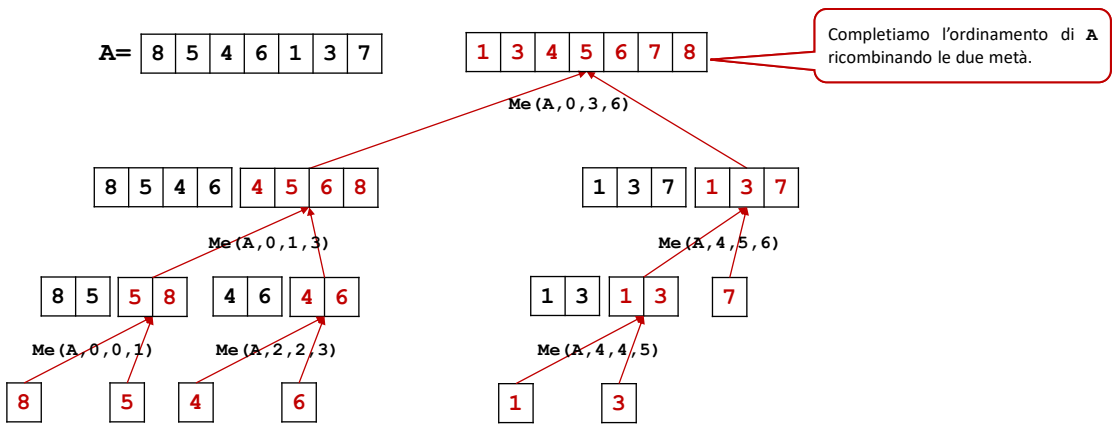
22

MergeSort



23

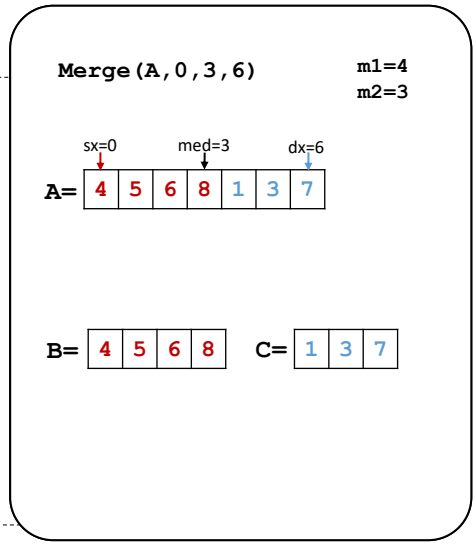
MergeSort



24

La procedura **Merge**

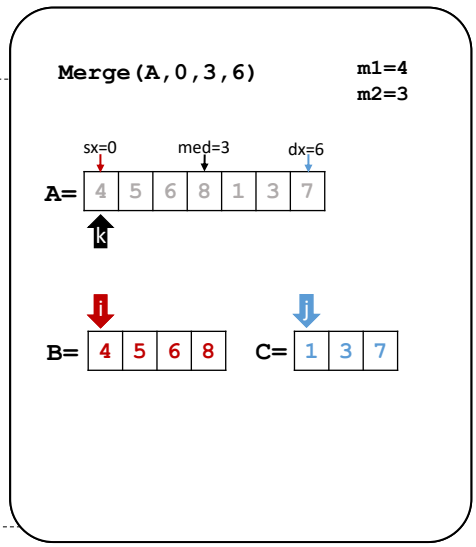
```
Merge(A,sx,med,dx) {
        m1=med-sx+1;
        m2=dx-med;
        B[0..m1-1]=A[sx..med];
        C[0..m2-1]=A[med+1..dx];
        i=j=0; k=sx;
        while(i<m1 && j<m2) {
                if (B[i]<=C[j])
                        A[k++]=B[i++];
                else
                        A[k++]=C[j++];
        }
```



25

La procedura **Merge**

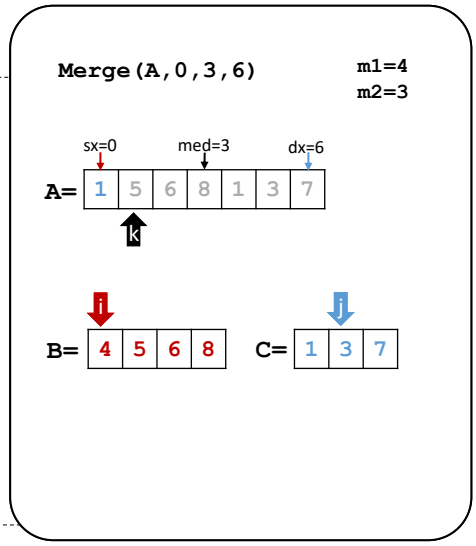
```
Merge(A,sx,med,dx) {
        m1=med-sx+1;
        m2=dx-med;
        B[0..m1-1]=A[sx..med];
        C[0..m2-1]=A[med+1..dx];
        i=j=0; k=sx;
        while(i<m1 && j<m2) {
                if (B[i]<=C[j])
                        A[k++]=B[i++];
                else
                        A[k++]=C[j++];
        }
```



26

La procedura **Merge**

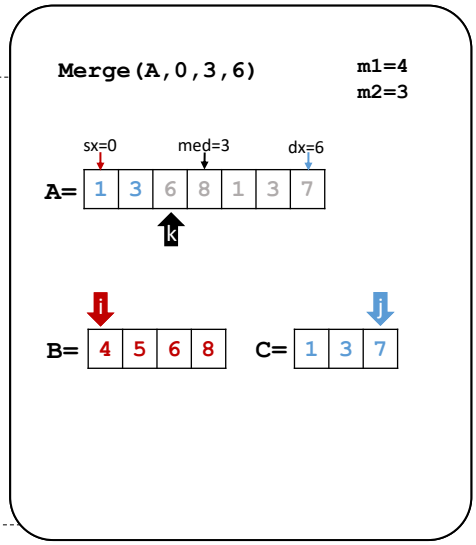
```
Merge(A,sx,med,dx) {
        m1=med-sx+1;
        m2=dx-med;
        B[0..m1-1]=A[sx..med];
        C[0..m2-1]=A[med+1..dx];
        i=j=0; k=sx;
        while(i<m1 && j<m2) {
                if (B[i]<=C[j])
                        A[k++]=B[i++];
                else
                        A[k++]=C[j++];
        }
```



27

La procedura **Merge**

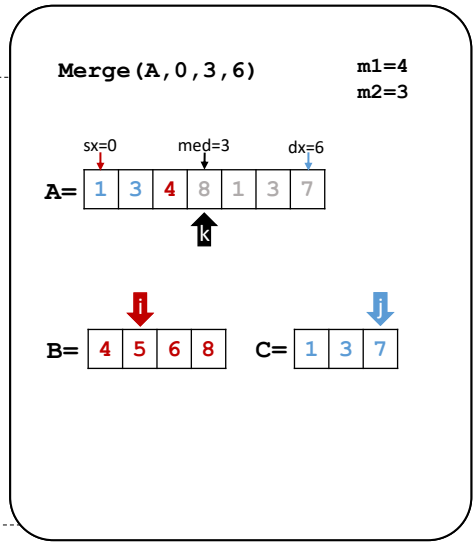
```
Merge(A,sx,med,dx) {
        m1=med-sx+1;
        m2=dx-med;
        B[0..m1-1]=A[sx..med];
        C[0..m2-1]=A[med+1..dx];
        i=j=0; k=sx;
        while(i<m1 && j<m2) {
                if (B[i]<=C[j])
                        A[k++]=B[i++];
                else
                        A[k++]=C[j++];
        }
```



28

La procedura **Merge**

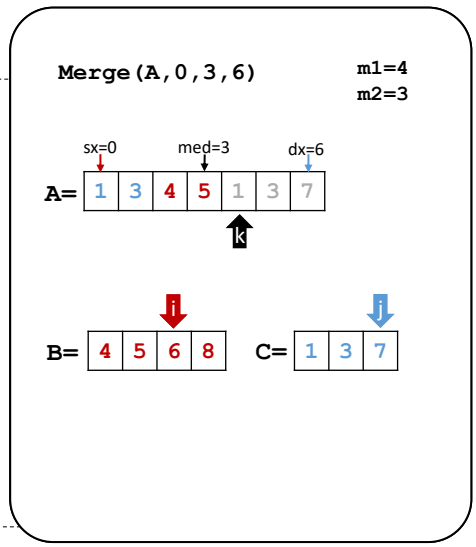
```
Merge(A,sx,med,dx) {
        m1=med-sx+1;
        m2=dx-med;
        B[0..m1-1]=A[sx..med];
        C[0..m2-1]=A[med+1..dx];
        i=j=0; k=sx;
        while(i<m1 && j<m2) {
                if (B[i]<=C[j])
                        A[k++]=B[i++];
                else
                        A[k++]=C[j++];
        }
```



29

La procedura **Merge**

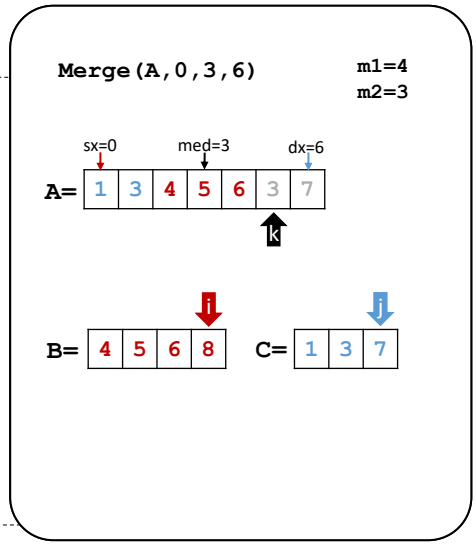
```
Merge(A,sx,med,dx) {
        m1=med-sx+1;
        m2=dx-med;
        B[0..m1-1]=A[sx..med];
        C[0..m2-1]=A[med+1..dx];
        i=j=0; k=sx;
        while(i<m1 && j<m2) {
                if (B[i]<=C[j])
                        A[k++]=B[i++];
                else
                        A[k++]=C[j++];
        }
```



30

La procedura **Merge**

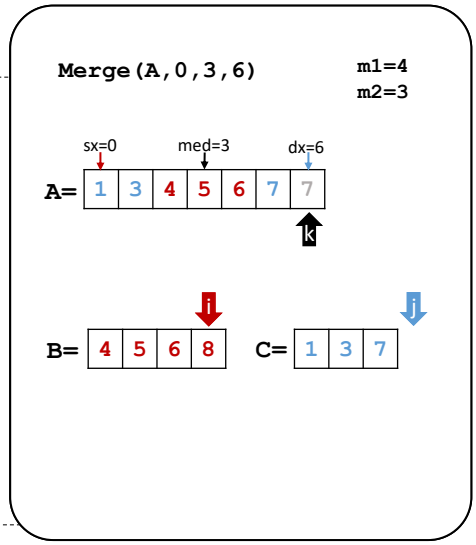
```
Merge(A,sx,med,dx) {
        m1=med-sx+1;
        m2=dx-med;
        B[0..m1-1]=A[sx..med];
        C[0..m2-1]=A[med+1..dx];
        i=j=0; k=sx;
        while(i<m1 && j<m2) {
                if (B[i]<=C[j])
                        A[k++]=B[i++];
                else
                        A[k++]=C[j++];
        }
```



31

La procedura **Merge**

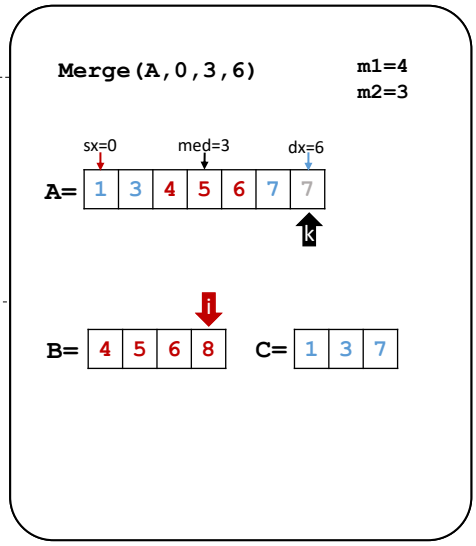
```
Merge(A,sx,med,dx) {
        m1=med-sx+1;
        m2=dx-med;
        B[0..m1-1]=A[sx..med];
        C[0..m2-1]=A[med+1..dx];
        i=j=0; k=sx;
        while(i<m1 && j<m2) {
                if (B[i]<=C[j])
                        A[k++]=B[i++];
                else
                        A[k++]=C[j++];
        }
```



32

La procedura **Merge**

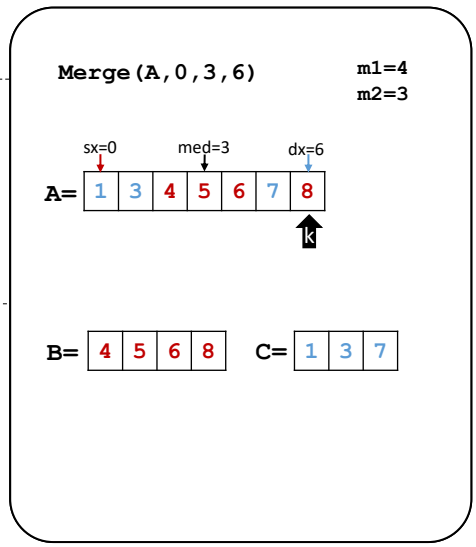
```
…
        if (i<m1)
                A[k..dx]=B[i..m1];
        else
                A[k..dx]=C[j..m2];
}
```



33

La procedura **Merge**

```
…
        if (i<m1)
                A[k..dx]=B[i..m1];
        else
                A[k..dx]=C[j..m2];
}
```



34

Analisi di **MergeSort**

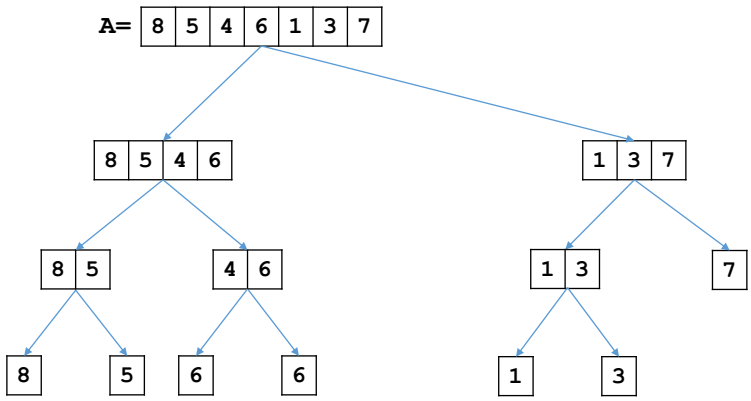
- Il costo di **MergeSort** è espresso da una *relazione di ricorrenza*

$$T(n) = \begin{cases} 1 & \text{se } n=1 \\ 2T\left(\frac{n}{2}\right) + f(n) & \text{se } n>1 \end{cases}$$

Dove $f(n)$ rappresenta il costo della sola **Merge**…

35

Analisi di **MergeSort**

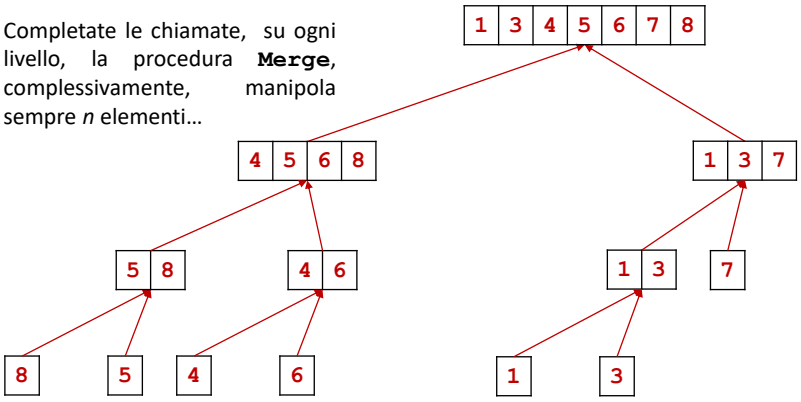


A ogni livello, una chiamata a **MergeSort** su *n* elementi, produce due chiamate a **MergeSort** su *n/2* elementi. …

36

MergeSort

Completate le chiamate, su ogni livello, la procedura **Merge**, complessivamente, manipola sempre *n* elementi…



Quindi, in pratica, $f(n) = O(n)$

37

Analisi di **MergeSort**

- Il costo di **MergeSort** è espresso da una *equazione di ricorrenza*

$$T(n) = \begin{cases} 1 & \text{se } n=1 \\ 2T\left(\frac{n}{2}\right) + f(n) & \text{se } n>1 \end{cases}$$

Dove $f(n)$ rappresenta il costo della sola **Merge**…

Applicando il c.d. *Master Theorem* per la soluzione di questo tipo di ricorrenze, si dimostra che:

$$T(n) = O(n \log_2 n)$$

38

Algoritmi di ordinamento *(per confronti)*

Algoritmo	*Caso migliore*	*Caso medio*	*Caso peggiore*
`SelectionSort`	$O(n^2)$	$O(n^2)$	$O(n^2)$
`InsertionSort`	$O(n)$	$O(n^2)$	$O(n^2)$
`BubbleSort`	$O(n)$	$O(n^2)$	$O(n^2)$
`MergeSort`	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$
`Quick Sort`	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n^2)$
`HeapSort`	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$

39

Algoritmi di ordinamento *(per confronti)*

- Si dimostra che Il limite inferiore alla complessità del problema dell'ordinamento se risolto esclusivamente mediante confronti (e quindi senza fare assunzioni sui dati da ordinare) è proprio $O(n \log_2 n)$
  - L'algoritmo **`SelectionSort`** è molto semplice ma piuttosto inefficiente, infatti ordina in $O(n^2)$ anche nel suo caso migliore.

    `BubbleSort` e **`InsertSort`**, lavorano in tempo polinomiale, ma in alcuni casi, evitano confronti «ridondanti».
  - L'algoritmo **`MergeSort`** (come pure **`HeapSort`**) è *ottimale*
  - **`QuickSort`** è un algoritmo mediamente molto efficiente, tuttavia nel caso peggiore le sue prestazione degradano in una complessità *polinomiale*

40