

Esercizio

- È dato un vettore di dimensione $N+k$ contenente N numeri interi (N può essere anche 0), ordinati in senso non decrescente
- Si suppone di ricevere uno alla volta k interi e di inserirli nel vettore, mantenendo l'ordinamento del vettore ad ogni passo di inserimento

Esercizio

- Per effettuare un inserimento occorre conoscere:
 - Il vettore
 - Il valore da inserire
 - La dimensione fisica del vettore = $N+k$
 - La dimensione logica del vettore (quanti elementi presenti) = N

- Interfaccia

```
typedef enum {false, true} Boolean;
Boolean insert(int vector[], int el, int dim,
              int *elCount);
```

Esercizio

```
Boolean insert(int intArray[], int el, int dim, int *elCount)
{
    int i = 0, j;
    Boolean found = false;
    if (*elCount < dim)
    {
        while (i < *elCount && !found)
        {
            if (el <= intArray[i])
                found = true;
            else
                i++;
        }
        if (found)
            for(j = *elCount; j >= i; j--)
                intArray[j] = intArray[j - 1]; /* shift */
        intArray[i] = el;
        (*elCount)++;
        return true;
    }
    return false;
}
```

Ricerca della posizione di inserimento...

...eventuale shift...

...inserimento e notifica del successo

3

Considerazioni

- Il tipo booleano può tornare utile e sarebbe bello poterlo inserire in un header file da includere al bisogno
- Il tipo booleano può servire in più punti del programma ma all'interno di una stessa applicazione non è possibile dichiarare più volte lo stesso tipo (typedef)
 - → includere più volte lo stesso header file con la stessa dichiarazione di tipo...
- Come risolvere il problema?

Il preprocessore!

- #Se il simbolo **BOOLEAN** non è definito
 - #Lo si definisce
 - Si dichiara il tipo **Boolean**

```
#ifndef BOOLEAN
#define BOOLEAN

typedef enum {false, true} Boolean;

#endif
```

Possibile includere l'header file ovunque necessario avendo la certezza che la definizione del tipo verrà processata una sola volta → non più necessario fare supposizioni sul fatto che qualcun altro nello stesso programma abbia già incluso l'header

Domanda al volo

Perché usando

```
typedef enum {true, false} Boolean;
```

anziché

```
typedef enum {false, true} Boolean;
```

il precedente algoritmo di inserimento ordinato non funzionerebbe più?

Algoritmi di Ordinamento

- Occorre avere a che fare con array di elementi sul cui tipo sia definita una **relazione d'ordine totale**
 - Ad esempio, tipo `float`
 - Più in generale sui tipi scalari
- Per semplicità si suppongano effettuate le dichiarazioni che seguono:

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

Algoritmi di Ordinamento

- Si supponga definita la funzione che segue

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

Algoritmi di Ordinamento

- Affinché il tipo di dato sia veramente “completo” e “astratto” occorrerebbero anche le funzioni...

...di confronto

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

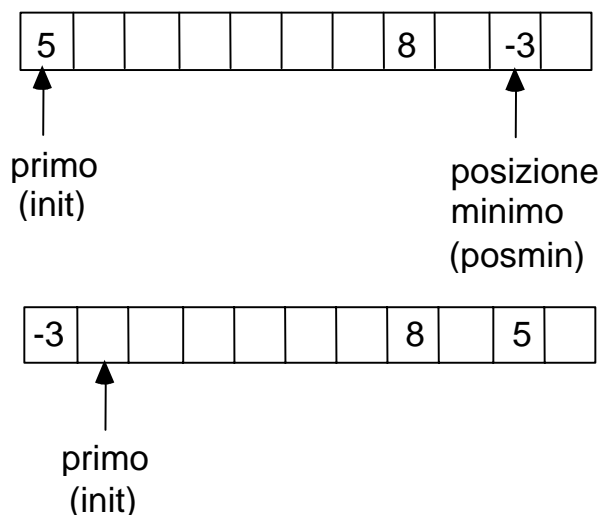
...e di assegnamento

```
void assign(ELEMENT *lvalue, ELEMENT rvalue);
```

Per evitare che il codice si appesantisca troppo, assumiamo che il tipo ELEMENT sia “elementare” e che il C sappia come confrontarlo e assegnarlo

Naive Sort

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



Naive Sort – Iterativo

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

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

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

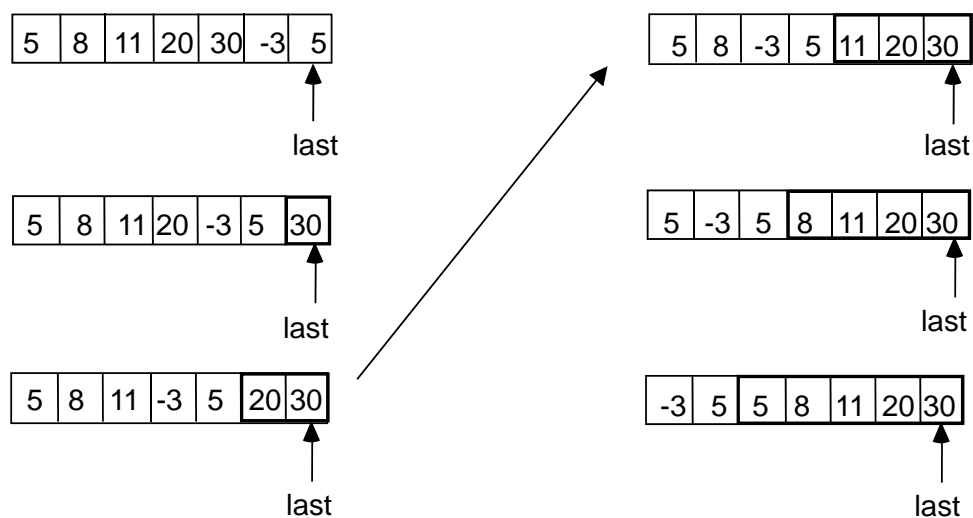
Naive Sort - Osservazioni

- Si supponga di avere a che fare con un array di dimensione N
- La ricerca del minimo si ripete N-1 volte (ciclo for esterno)
- Per trovare il minimo, l'istruzione di confronto (che si può considerare dominante) viene eseguita:
n-1 n-2 ... 3 2 1 volte
1 2 ... n-3 n-2 n-1 passo
- $S(i=1..n-1) i = n*(n-1)/2 = O(n^2)$
- Questo risultato è **indipendente dai valori di ingresso**
- Comunque si eseguono $O(n^2)$ confronti anche se il vettore è già ordinato

Bubble Sort

- Ordinamento a “*bolla*”
- Si basa sul fatto che esiste un **ordinamento totale** sugli elementi del vettore: dati due elementi adiacenti $a[i]$ e $a[i+1]$, se non rispettano l'ordinamento vengono scambiati

Bubble Sort



Bubble Sort

- Pseudo codifica – versione ricorsiva
 - <per tutte le coppie di elementi adiacenti dell'array corrente **a** esegui:>
 - <se $a[i] > a[i+1]$ allora scambiali>
 - <se l'array non è ordinato...>
 - <ordina l'array ottenuto da **a** eliminando l'ultimo elemento>
- Pseudo codifica – versione iterativa
 - do**
 - <per tutte le coppie di elementi adiacenti dell'array corrente **a** esegui:>
 - <se $a[i] > a[i+1]$ allora scambiali>
 - <imposta come array corrente l'array **a** tolto l'ultimo elemento>
 - while** (<il vettore **a** è ordinato>)

Bubble Sort

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

Bubble Sort - Ricorsivo

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

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

19

Bubble Sort – Iterativo

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

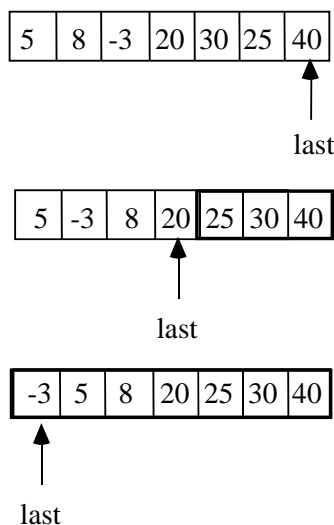
    do
    {
        swapped = false;
        for (i = 0; i < dim - 1; i++)
        {
            if (a[i] > a[i + 1])
            {
                swapped = true;
                swap(&a[i], &a[i + 1]);
            }
        }
        dim--;
    }
    while (swapped);
}
```

20

Bubble Sort Ottimizzato

- Si tiene traccia della posizione in cui è stato effettuato l'ultimo scambio
 - Nelle posizioni successive alla posizione dell'ultimo scambio l'array è ordinato!
- Ad ogni iterazione si esclude la parte finale del vettore già ordinata (e non solo l'ultimo elemento)

Bubble Sort Ottimizzato



- **last** rappresenta la posizione in cui è avvenuto l'ultimo scambio
- All'inizio dell'algoritmo viene inizializzata a zero
- La condizione di terminazione è **last == 0** → l'array è ordinato oppure l'ultimo scambio è avvenuto all'inizio dell'array

Bubble Sort Ottimizzato - Ricorsivo

```
void bubbleSortOptR(ARRAY a, int dim)
{
    int i, last;
    last = 0;
    for (i = 0; i < dim-1; i++)
    {
        if (a[i] > a[i + 1])
        {
            swap(&a[i], &a[i + 1]);
            last = i;
        }
    }
    if (last > 0)
        bubbleSortOptR(a, last + 1);
}
```

Bubble Sort Ottimizzato - Iterativo

```
void bubbleSortOpt(ARRAY a, int dim)
{
    int i, limit, last;
    limit = dim - 1;
    while (limit > 0)
    {
        last = 0;
        for (i = 0; i < limit; i++)
        {
            if (a[i] > a[i + 1])
            {
                swap(&a[i], &a[i + 1]);
                last = i;
            }
        }
        limit = last;
    }
}
```

limit serve perché last viene modificato all'interno del ciclo...

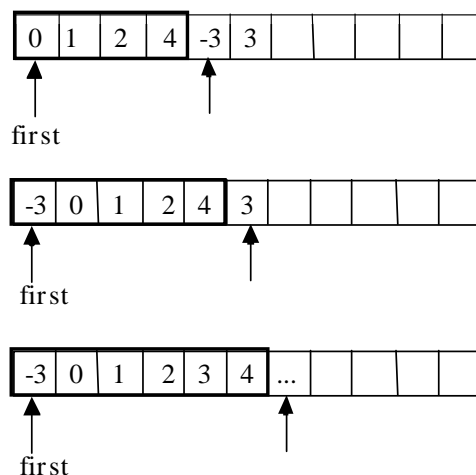
...alla fine del ciclo, last viene assegnato a limit

Bubble Sort - Osservazioni

- Non sono sempre necessarie $n-1$ iterazioni
 - se non avviene alcuno scambio, l'algoritmo termina
- **Dipende dai valori dei dati in ingresso**
- **Caso migliore:** array già ordinato
 - Una sola iterazione, con $n-1$ confronti e nessuno scambio
- **Caso peggiore:** array ordinato in senso decrescente
 - Al passo di iterazione i , $(n-i)$ confronti e $(n-i)$ scambi
→ $S = (n-1)*n$
 - L'ordine di grandezza del numero di scambi è $O(n^2)$, in modo analogo a quello di Naive Sort

Insertion Sort

- L'ordinamento è ottenuto costruendo un sotto-array ordinato a partire dalla prima componente
- In questo sotto-array gli elementi sono inseriti ordinatamente e l'inserimento è ottenuto attraverso "shift" a destra dei restanti elementi del vettore



Insertion Sort

■ Pseudo codifica

- <per ogni elemento dell'array (elemento di scansione) a partire dal secondo, esegui:>
 - <considera il sotto-array che va dal primo elemento all'elemento di scansione (escluso)>
 - <trova la posizione di inserimento dell'elemento di scansione nel sotto-array>
 - <sposta tutti gli elementi in avanti dalla posizione di inserimento all'elemento precedente l'elemento di scansione>
 - <copia l'elemento di scansione nella posizione trovata>

Insertion Sort

```
void insertionSort (ARRAY a, int dim)
{
    int scanIdx;
    for (scanIdx = 1; scanIdx < dim; scanIdx++)
    {
        int subIdx, pos;
        ELEMENT el;
        Boolean found = false;
        el = a[scanIdx];
        for (subIdx = 0; subIdx < scanIdx && !found; subIdx++)
            if (el <= a[subIdx]) /* find pos */
            {
                found = true;
                pos = subIdx;
            }
        if (found) /* shift */
            for(subIdx = scanIdx; subIdx > pos; subIdx--)
                a[subIdx] = a[subIdx-1];
        else
            pos = subIdx;
        a[pos] = el; /* insert */
    }
}
```

Insertion Sort – Ottimizzazione

- L'implementazione è semplice (e chiara?) ma il codice non è affatto efficiente
 - **Prima** si fa un ciclo per trovare la posizione di inserimento
 - **Poi** si fa un altro ciclo per fare posto all'elemento spostando in avanti gli elementi
- Si può fare tutto con un unico ciclo?

<scorri l'array partendo dal secondo elemento e, per ogni elemento...>
<vai indietro per cercare la corretta posizione di inserimento>
<contemporaneamente, sposta gli elementi in avanti>
<trovata la posizione inserisci l'elemento>

Insertion Sort – Ottimizzato

```
void insertionSortOpt(ARRAY a, int dim)
```

```
{
```

```
    int i, j;
```

```
    ELEMENT tmp;
```

Parte dal secondo elemento

Mette in una variabile temporanea

```
    for (i = 1; i < dim; i++)
```

```
    {
```

```
        tmp = a[i];
```

Finché non trova la posizione giusta, va indietro e...

```
        for (j = i - 1; j >= 0 && a[j] > tmp; j--)
```

```
            a[j + 1] = a[j];
```

...sposta gli elementi in avanti

```
        a[j + 1] = tmp;
```

```
    }
```

```
}
```

Trovata la posizione, sistema l'elemento

Insertion Sort – Osservazioni

- **Caso migliore:** array già ordinato
→ $n-1$ confronti
- **Caso peggiore:** considerazioni simili agli algoritmi precedenti
- Quindi, non introduce guadagni prestazionali significativi...

Shell Sort

- Lo Shell sort è una **estensione dell'insertion sort**, tenendo presenti due osservazioni:
 - L'insertion sort è efficiente se array di ingresso è già abbastanza ordinato
 - L'insertion sort è inefficiente, generalmente, in quanto muove i valori di una sola posizione per volta
- Lo Shell sort è simile all'insertion sort, ma funziona spostando i valori di più posizioni per volta (il passo è più grande di uno); gradualmente viene diminuita la dimensione del passo sino ad arrivare ad uno
- Alla fine, lo Shell sort esegue un insertion sort, ma per allora i dati saranno già piuttosto ordinati → efficienza
- Consideriamo un valore piccolo posizionato inizialmente all'estremità errata di un array dati di lunghezza n . Con lo Shell sort, si muoveranno i valori usando passi di grosse dimensioni, cosicché un valore piccolo andrà velocemente nella sua posizione finale con pochi confronti e scambi

Shell Sort

- **Algoritmo:**
 - sistema la sequenza dei dati in un array bidimensionale (con un numero h di colonne)
 - ordina i valori presenti all'interno di ciascuna colonna dell'array
 - ripeti dal punto 1 con un diverso numero h (minore del precedente) fino a portare h ad 1
- **Alla fine la sequenza dei dati viene parzialmente ordinata**
 - La procedura viene eseguita ripetutamente, ogni volta con un array più piccolo, cioè, con un numero di colonne h più basso
 - Nell'ultima passata, l'array è composto da una singola colonna ($h=1$) trasformando di fatto questo ultimo giro in un insertion sort puro e semplice
 - Ad ogni passata i dati diventano sempre più ordinati, finché, durante l'ultima lo diventano del tutto
 - Comunque, il numero di operazioni di ordinamento necessarie in ciascuna passata è limitato, a causa dell'ordinamento parziale ottenuto nelle passate precedenti

33

Shell Sort – visione implementativa

- Dovendo lavorare in un array, occorre “simulare” di lavorare con una matrice
- Vengono esaminate coppie di elementi che si trovano ad una distanza prefissata *gap*...
- ...quindi si applica l'insertion sort agli array fittizi formati dagli elementi a *gap* distanza l'uno dall'altro
- Inizialmente *gap* è pari alla metà della dimensione dell'array, poi decresce fino ad 1 (in questo caso → insertion sort)

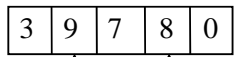
Shell Sort

Array di 5 elementi \rightarrow **gap** = 2

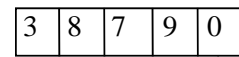
Algoritmo insertion sort con **gap** = 2



3 in una var. temp.; $7 >$ var. temp., viene portato avanti; raggiunto inizio array \rightarrow var. temp. copiata all'inizio del sotto-array (posto 0)



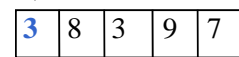
8 in una var. temp.; $9 >$ var. temp., viene portato avanti; raggiunto inizio array \rightarrow var. temp. copiata all'inizio del sotto-array (posto 1)



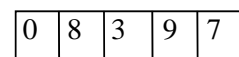
0 in una var. temp.; $7 >$ var. temp., viene portato avanti; non raggiunto inizio array \rightarrow si retropropaga...



... $3 >$ var. temp., viene portato avanti...



...raggiunto inizio array \rightarrow var. temp copiata all'inizio del sotto-array (posto 0)

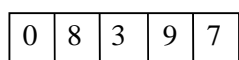


Terminato l'algoritmo con con **gap** = 2, i due sotto-array sono ordinati \rightarrow Si procede con **gap** = 1

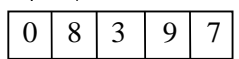
Shell Sort

Array di 5 elementi \rightarrow **gap** = 2

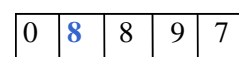
Algoritmo insertion sort con **gap** = 2



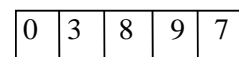
8 in una var. temp.; $0 <$ var. temp., si procede



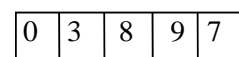
3 in una var. temp.; $8 >$ var. temp., viene portato avanti; non raggiunto inizio array \rightarrow si retropropaga...



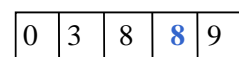
$0 <$ var. temp \rightarrow si termina la retropropagazione e si copia var. temp al posto dell'ultimo elemento portato avanti



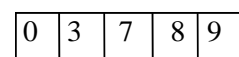
9 in una var. temp.; $8 <$ var. temp., di procede



7 in una var. temp.; $9 >$ var. temp., viene portato avanti; non raggiunto inizio array \rightarrow si retropropaga...



$8 >$ var. temp., viene portato avanti \rightarrow si retropropaga...



$3 <$ var. temp., si interrompe la retropropagazione e si copia var. temp. Al posto dell'ultimo elemento portato avanti.

Array ORDINATO!!!

Shell Sort

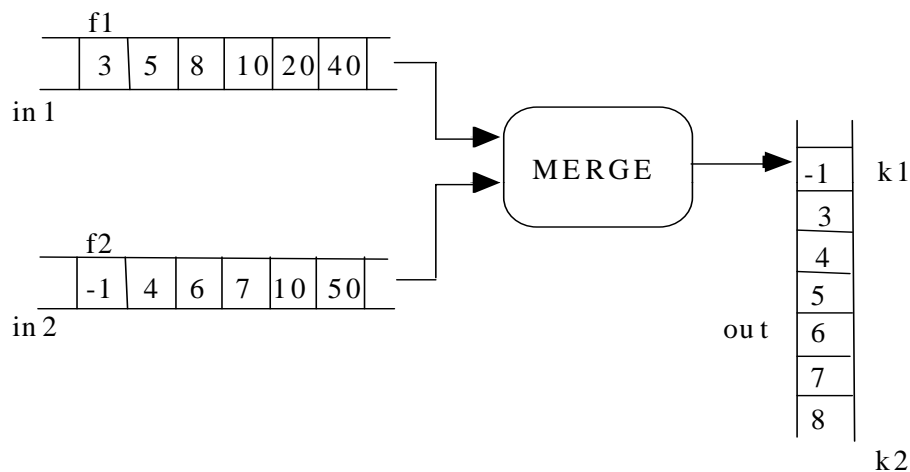
```
void shellSort(ARRAY a, int dim)
{
    int i, j, gap;
    ELEMENT tmp;
    for (gap = dim / 2; gap > 0; gap /= 2)
    {
        for (i = gap; i < dim; i++)
        {
            tmp = a[i];
            for (j = i - gap; j >= 0 && a[j] > tmp; j -= gap)
                a[j + gap] = a[j];
            a[j + gap] = tmp;
        }
    }
}
```

37

Merge Sort

- Ordinamento *per fusione*
- Utilizza un algoritmo di Merge
 - Dati due vettori \mathbf{x} , \mathbf{y} con m componenti ciascuno e ordinati in ordine crescente, produrre un unico vettore \mathbf{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 $in1[i] \leq in2[j]$, $out[k] = in1[i]$ (scrivi nella componente corrente del vettore *out* $in1[i]$); altrimenti, $out[k]=in2[j]$

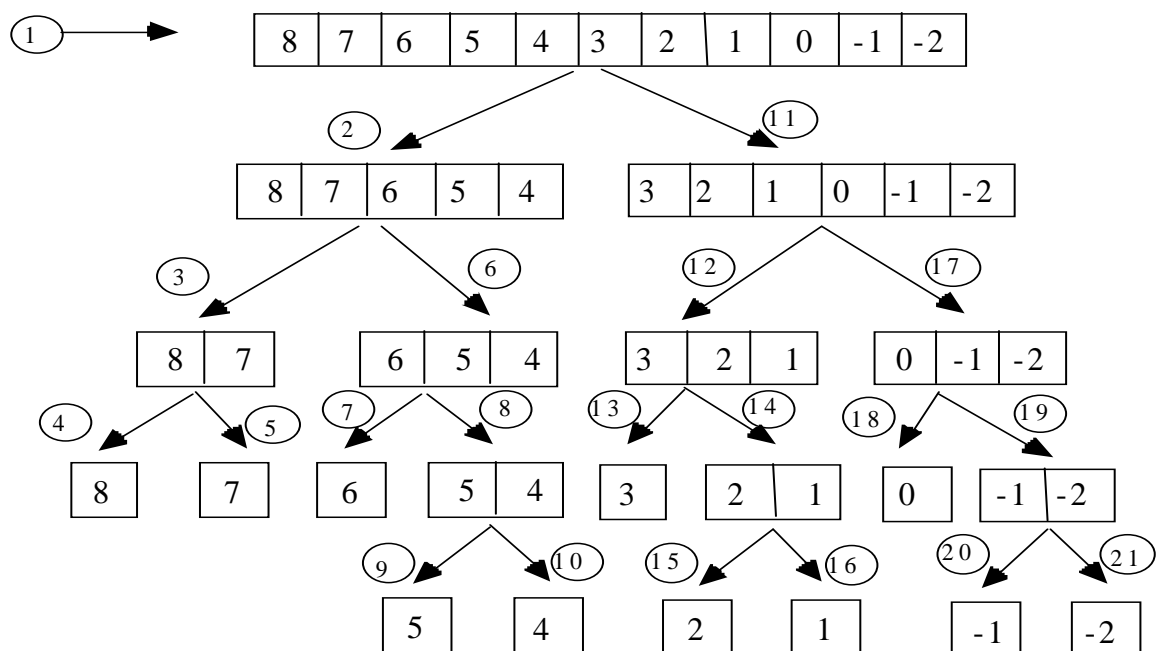
Algoritmo di Merge

- Indici *i*, *j* per scandire *in1* e *in2*, indice *k* per scrivere su *out*
- Si confrontano $in1[i]$ e $in2[j]$:
 - se $in1[i] \leq in2[j]$, scrive $in1[i]$ nella componente *k*-esima di *out* (incrementa *i*, *k*)
 - altrimenti, scrive $in2[j]$ nella componente *k*-esima di *out* (incrementa *j*, *k*)
- Se la scansione di uno dei vettori è arrivata all'ultima componente, si copiano i rimanenti elementi dell'altro nel vettore *out*

Merge Sort

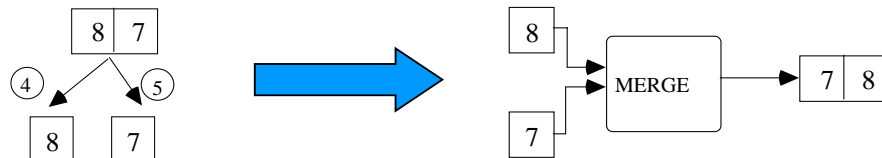
- È un algoritmo per sua natura **ricorsivo**
- Il vettore di ingresso viene diviso in due sotto-vettori sui quali si richiama il merge sort
- Quando ciascun sotto-vettore è ordinato, i due vengono “fusi” attraverso la procedura di merge
- **Si può dimostrare che Merge Sort è il più efficiente algoritmo di ordinamento**
- Il numero di “operazioni” effettuato è una funzione che si comporta come $n \cdot \log_2 n$

Merge Sort



Merge Sort

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



Merge Sort

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

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

...mantiene uniformi le signature delle funzioni di ordinamento...

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

Merge Sort

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

Merge Sort

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

Merge Sort - Costo

- Il costo dell'operazione di Merge è proporzionale alla lunghezza degli array in ingresso $\rightarrow O(n)$
- La funzione Merge Sort richiama se stessa per due volte sulla metà dell'array in ingresso \rightarrow 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

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

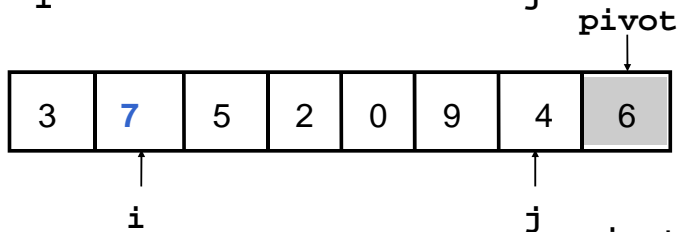
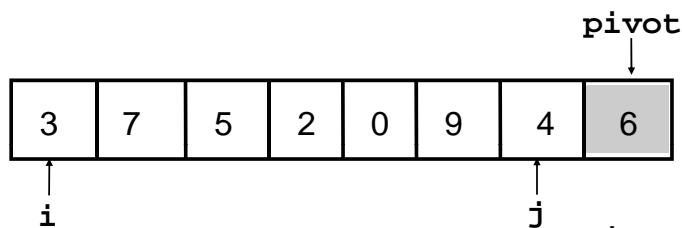
Quick Sort - Algoritmo

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

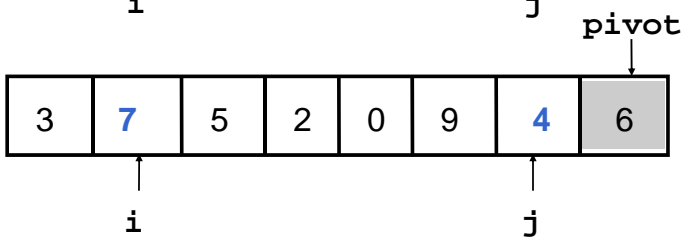
Quick Sort – Algoritmo

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

Quick Sort

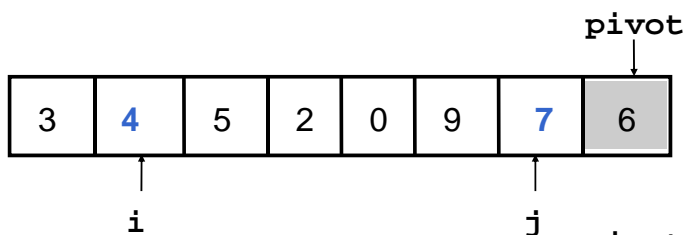


Scansione in avanti

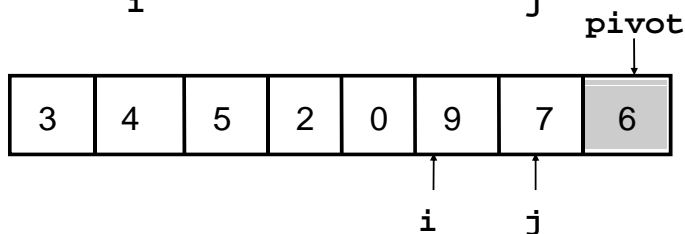


Scansione all'indietro

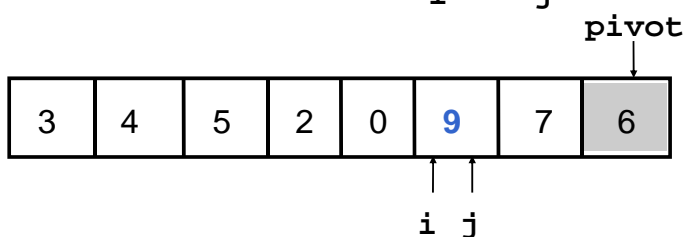
Quick Sort



Scambio

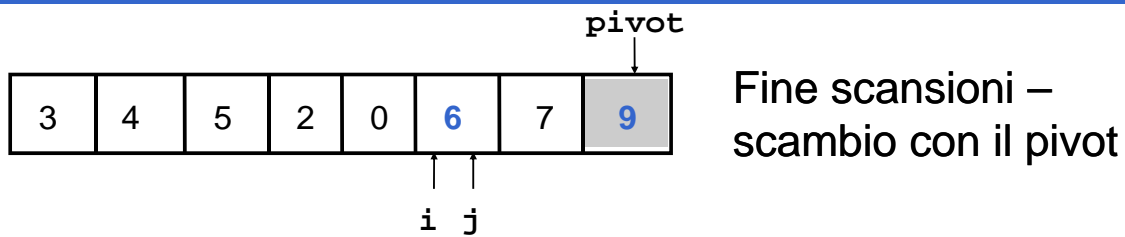


Scansione in avanti



Scansione all'indietro

Quick Sort



- Il pivot è nella posizione definitiva
- Ripetere il procedimento sui due sotto-array
 - $a[0, i - 1]$
 - $a[i + 1, \text{dim} - 1]$

Quick Sort

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

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

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

Quick Sort

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

continua...

Quick Sort

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

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

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

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

Quick Sort

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

Proposta di esercizio

- Sarebbe interessante vedere come i vari algoritmi si comportano con lo stesso array da ordinare
- Si potrebbe, per esempio, incrementare una variabile contatore globale all'interno della procedura **swap**
- Prima di invocare un ordinamento → azzeramento del contatore
- Dopo un ordinamento → stampa del contatore
- E gli algoritmi che non usano mai **swap**?

