ALGORITMI DI ORDINAMENTO

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

ALGORITMI DI ORDINAMENTO

Principali algoritmi di ordinamento:

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

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

==> Studio della Complessità: non lo tratteremo formalmente

Algoritmi di Ordinamento

- Si supponga definita la funzione che segue

```
void scambia(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

NAÏVE SORT

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

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

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

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

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

  */** ***invariante: v[n-1] contiene il massimo*** **/*

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

  `}`

NAÏVE SORT

Codifica

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

La dimensione dell'array cala di 1 a ogni iterazione

NAÏVE SORT

Codifica

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

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

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

NAÏVE SORT

Valutazione di complessità

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

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

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

BUBBLE SORT (ordinamento a bolle)

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

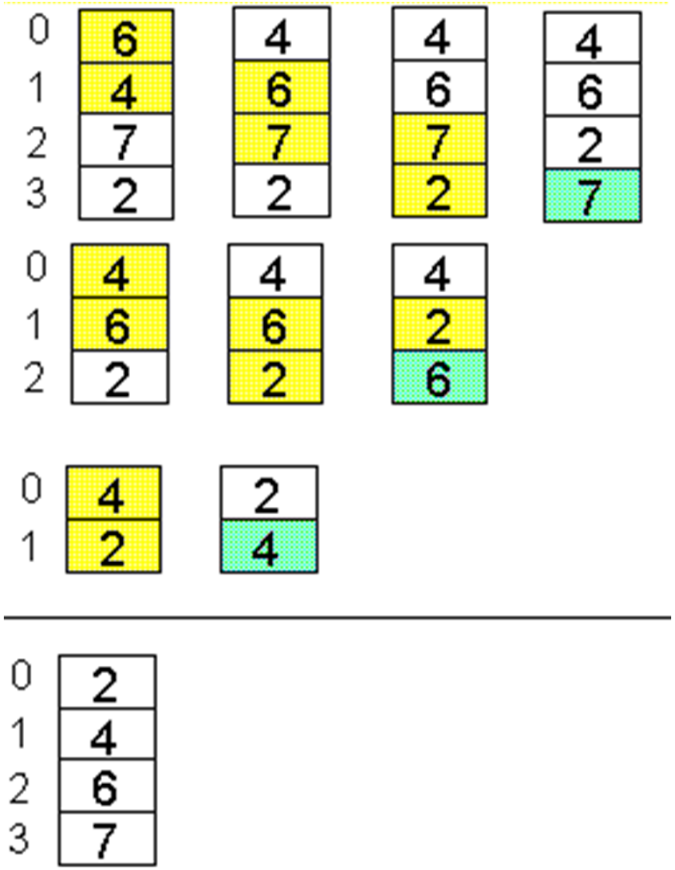
BUBBLE SORT

Codifica

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

BUBBLE SORT

Esempio



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

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

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

array ordinato

BUBBLE SORT

Valutazione di complessità

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

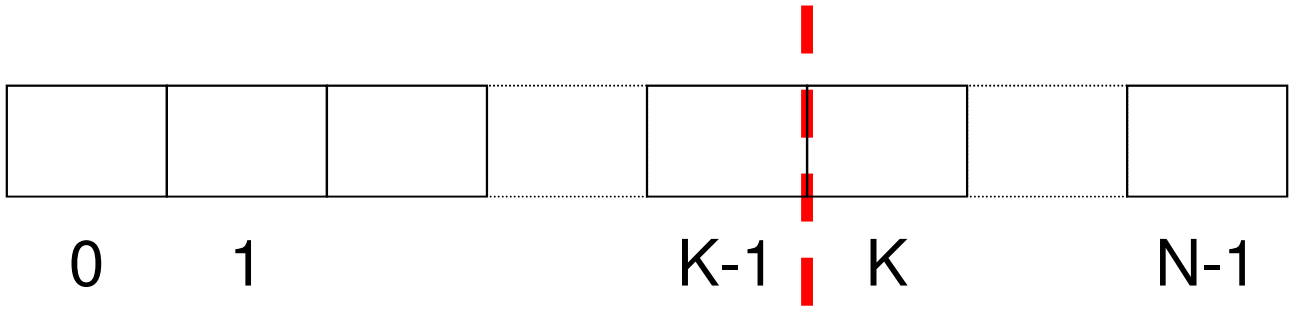
INSERT SORT

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

INSERT SORT

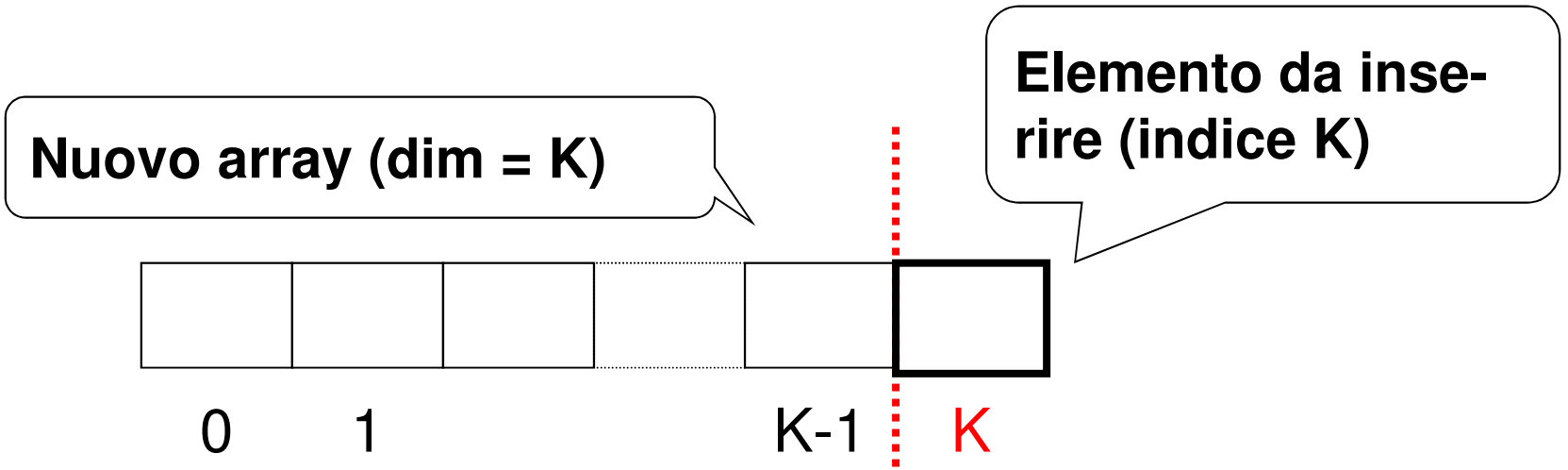
Scelta di progetto

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



INSERT SORT

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



INSERT SORT

Specifica

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

<considera l'elemento di indice k e inseriscilo nel nuovo vettore (da 0 a k) eventualmente spostando in avanti alcuni elementi>

Codifica

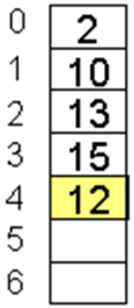
All'inizio (k=1) il nuovo array è la sola prima cella

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

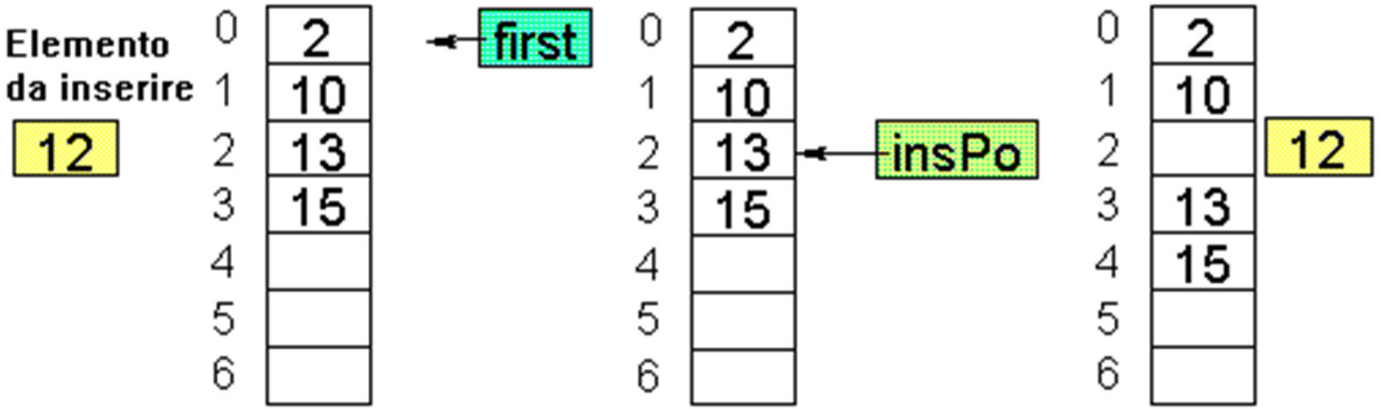
Al passo *k*, la demarcazione fra i due array è alla posizione k

INSERT SORT

Esempio



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



INSERT SORT

Specifica di insMinore()

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

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

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

<inserisci il nuovo elemento alla posizione prevista>

```
}
```

INSERT SORT

Codifica di insOrd()

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

Determina la posizione a cui inserire x

INSERT SORT

Esempio

INSERT SORT

Valutazione di complessità

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

QUICK e MERGE 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.***

QUICK e MERGE SORT

Algoritmo ricorsivo:

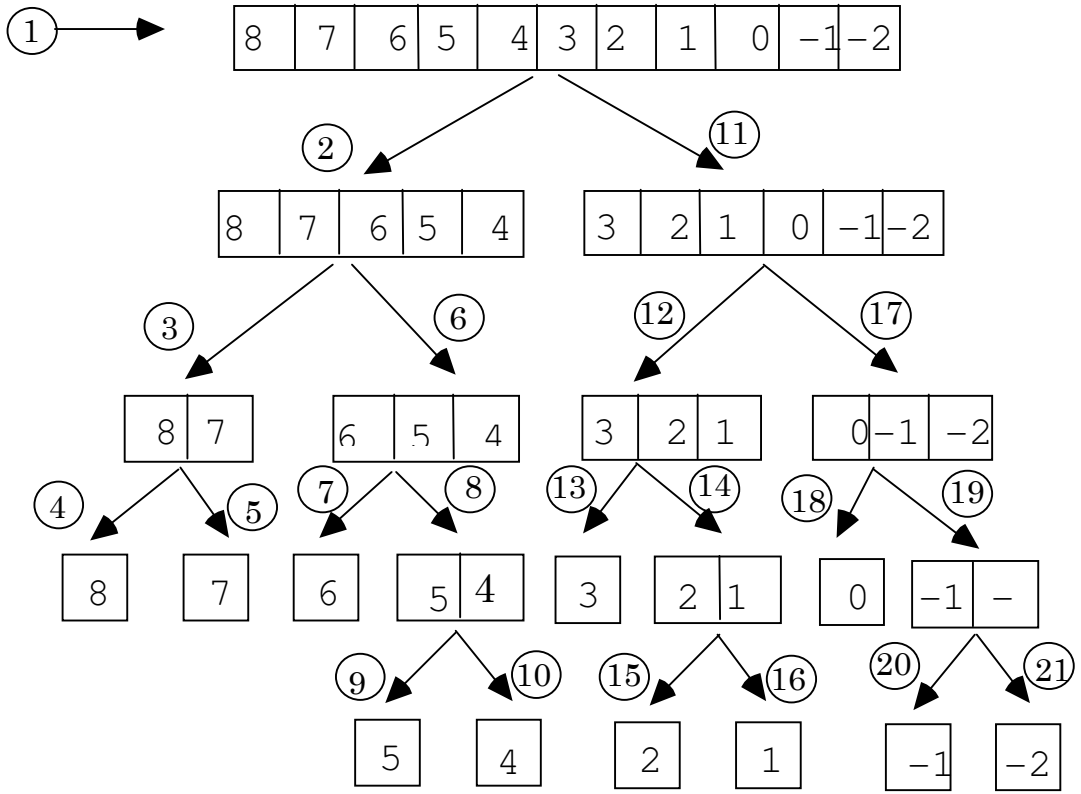
- **i due sub-array ripropongono un problema di ordinamento *in un caso più semplice* (array più corti)**
- **a forza di scomporre un array in sub-array, si giunge a un array di un solo elemento, che è già ordinato (*caso banale*).**

MERGE SORT

- **Produce *sempre* due sub-array di egual ampiezza**
- ***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 first, int last,
               int vout[]) {
  if (<array non vuoto>){
   <partiziona l'array in due metà>
   <richiama mergeSort ricorsivamente sui due sub-array,
    se non sono vuoti>
   <fondi in vout i due sub-array ordinati>
  }
}
```

E se avessi messo vout[] come variabile locale?

MERGE SORT

Codifica

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

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];
}
```

MERGE 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* $O(N \log_2 N)$**
- **MERGE SORT ottiene sempre il caso ottimo $O(N*\log_2 N)$**

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

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.

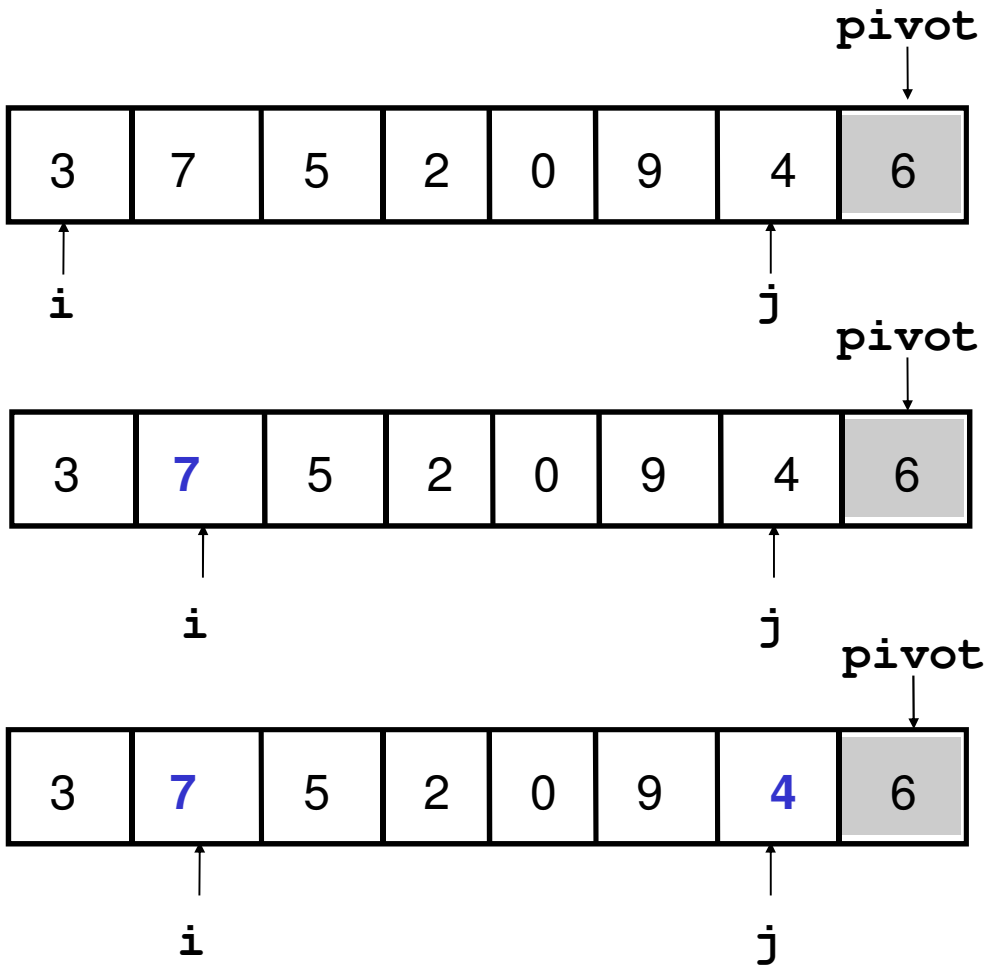
QUICK SORT

- 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

- **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!**

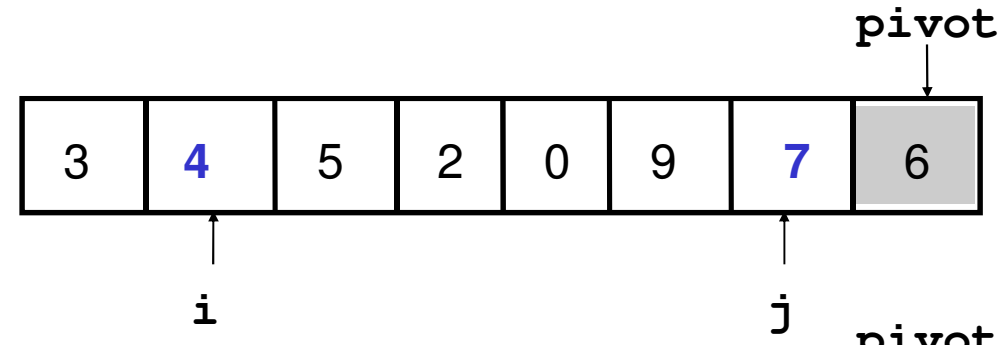
QUICK SORT



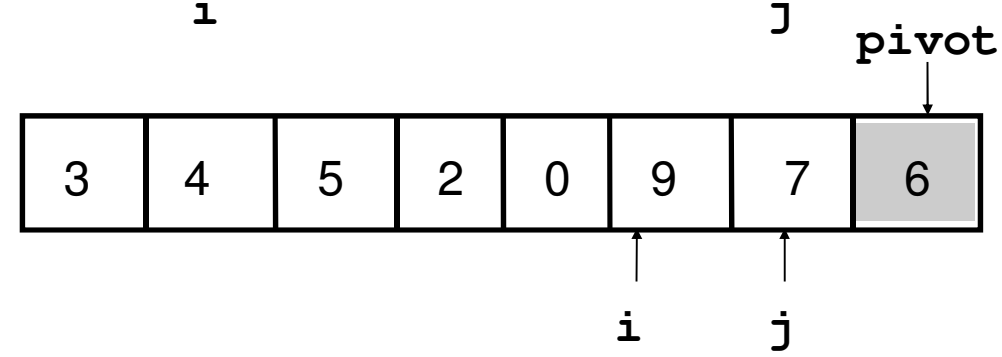
Scansione in avanti

Scansione all'indietro

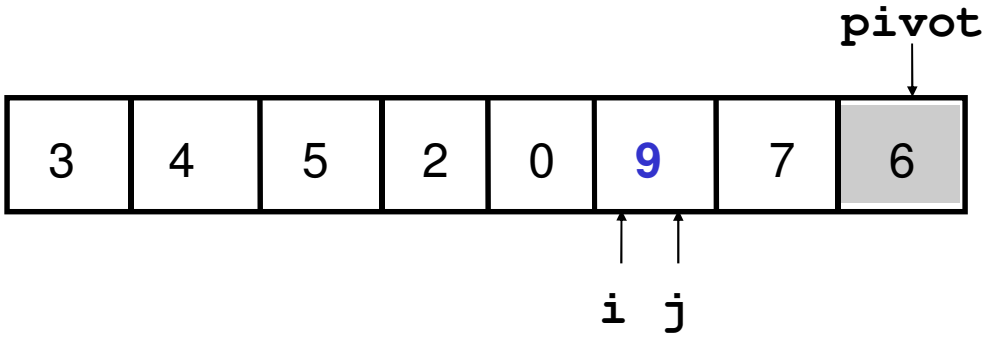
QUICK SORT



Scambio

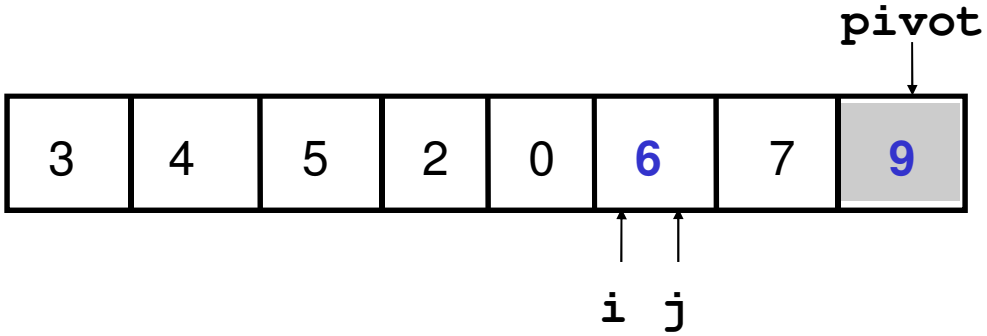


Scansione in avanti



Scansione all'indietro

QUICK SORT



Fine scansioni –
scambio con il pivot

- **Il pivot è nella posizione definitiva**
- **Ripetere il procedimento sui due sotto-array**
  - `a[0, i - 1]`
  - `a[i + 1, dim - 1]`

QUICK SORT

- **Si introduce una funzione che fa da interfaccia con i clienti e che invoca opportunamente la funzione ricorsiva**

```
void quickSortR(int a[], int iniz, int fine);

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

QUICK SORT

```
void quickSortR(int a[], int iniz, int fine)
{
    int i, j, iPivot, 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) scambia(&a[i], &a[j]);
        }
        while (i < j);
```

continua…

QUICK SORT

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

        if (i != iPivot && a[i] != a[iPivot])
        {
            scambia(&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

La complessità dipende dalla scelta del pivot:

- se il pivot è scelto male (uno dei due sub-array ha lunghezza zero), i confronti sono **$O(N^2)$**
- **se però il pivot è scelto bene (in modo da avere due sub-array di egual dimensione):**
  - **Numero globale di confronti: $O(N \log_2 N)$**
- Dunque:
- 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…)

ESPERIMENTI

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