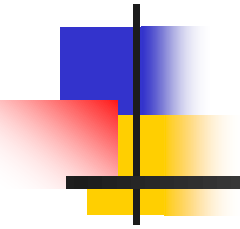
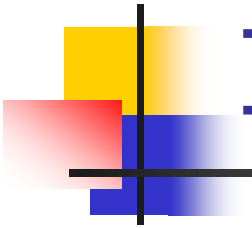


Algoritmi di Ordinamento

Fondamenti di Informatica

Prof. Ing. Salvatore Cavalieri



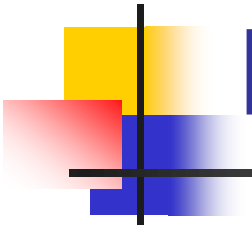
Introduzione

- Ordinare una sequenza di informazioni significa effettuare una permutazione in modo da rispettare una relazione d'ordine tra gli elementi della sequenza (per esempio minore o uguale ovvero non decrescente)

- Sulla sequenza ordinata diventa più semplice effettuare ricerche

Campi chiave

- L'ordinamento di una sequenza viene fatto scegliendo almeno un campo, definito **chiave**, che è quello utilizzato per la ricerca nella sequenza.
- È molto comune la ricerca per più di un campo (**chiavi multiple**)
- Le chiavi vanno scelte in modo da eliminare le possibili omonimie
- L'ordinamento è spesso effettuato su più di un campo: chiave primaria, secondaria etc.

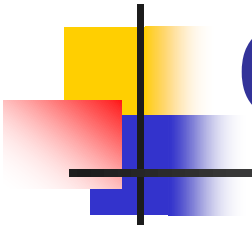


Esempio

1. Rossi Paolo 095456789
2. Rossi Carlo 095435612
3. Bianchi Agata 095353678

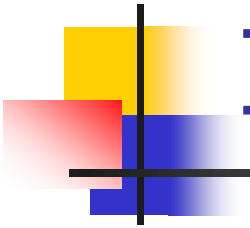
Volendo ordinare per Cognome e Nome in ordine crescente, si ottiene:

1. Bianchi Agata 095353678
2. Rossi Carlo 095435612
3. Rossi Paolo 095456789



Ordinamenti interni ed esterni

- Gli ordinamenti interni sono fatti su sequenze in memoria centrale
- Gli ordinamenti esterni sono fatti su sequenze in memoria di massa

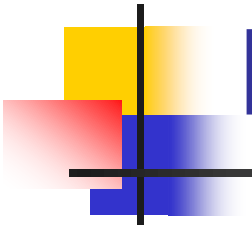


Ipotesi

- Consideriamo solo ordinamenti interni
- Supporremo che la sequenza di informazioni sia rappresentabile come vettore di n elementi, ovvero ogni elemento sia individuabile tramite un indice variabile da 0 a n-1.
- Gli elementi del vettore potranno essere tipi scalari (ad esempio interi o virgola mobile) o aggregati (struct)

Complessità computazionale

- È importante avere un indicatore di confronto tra i vari algoritmi possibili di ordinamento, indipendentemente dalla piattaforma hw/sw e dalla struttura dell'elemento informativo
- La complessità computazionale si basa sulla valutazione del numero di operazioni elementari necessarie (Confronti, Scambi)
- Si misura come funzione del numero n di elementi della sequenza: Notazione O (si dice «o grande»)
- Gli algoritmi di ordinamento interno si dividono in
  - Algoritmi semplici - complessità $O(n^2)$
  - Algoritmi evoluti - complessità $O(n*\log(n))$



Bubble Sort

- Si tratta di un algoritmo semplice.
- Il nome deriva dalla analogia dei successivi spostamenti che compiono gli elementi dalla posizione di partenza a quella ordinata simile alla risalita delle bolle di aria in un liquido.

Descrizione BubbleSort

- Si consideri un vettore di n elementi.
- Si supponga si vogliono ordinare gli elementi in ordine non decrescente
- Facciamo "risalire " gli elementi più piccoli dalle ultime posizioni alle posizioni iniziali del vettore
- Ad ogni ciclo vengono confrontati gli elementi del vettore, e se essi non sono nell'ordine desiderato, vengono scambiati
- Si utilizza una variabile "ordinato", che se 1 indica se il vettore è già ordinato e dunque l'algoritmo può terminare. Se la variabile è 0 allora il vettore non è ancora ordinato e l'algoritmo deve continuare

Codice C del Bubble Sort

```
1.  void scambia(tipobase v[], unsigned long i, unsigned long j)
2.  {
3.      tipobase tmp=v[i];
4.      v[i]=v[j];
5.      v[j]=tmp;
6.  }
```

Codice C del Bubble Sort (cont.)

```
1.  void BubbleSort(tipobase v[], unsigned long dim)
2.  {
3.      short ordinato=0;
4.      unsigned long i,j;
5.      for (j=0; j<dim-1 && !ordinato; j++) {
6.          ordinato=1;
7.          for (i=dim-1; i>j; i--)
8.                  if (v[i]<v[i-1]) {
                            scambia(v,i,i-1);
1.                          ordinato=0;
2.                  }
3.
4.      }
5.  }
```

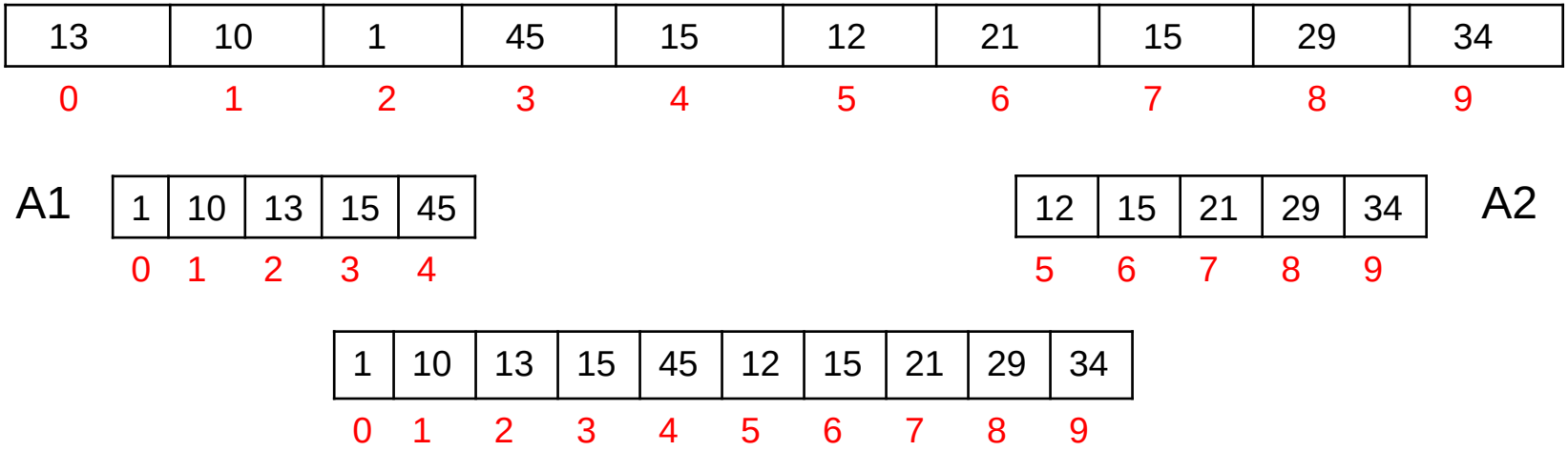
Attenzione !
Il < deve essere sostituito in dipendenza del tipo tipobase

Quick Sort

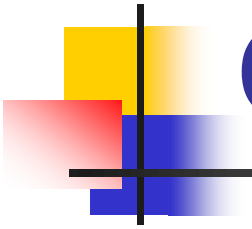
- Si tratta di un algoritmo evoluto che ha complessità computazionale **media** n*log(n) ed inoltre usa la ricorsione.
- Si consideri un vettore di n elementi e lo si ordini con algoritmi semplici, con un tempo di calcolo proporzionale a n^{2}.
- Si supponga, invece di dividere il vettore da ordinare in due sottovettori di n/2 elementi ciascuno e di ordinare le due metà separatamente.
- Applicando sempre gli algoritmi non evoluti, si avrebbe un tempo di calcolo pari a $(n/2)^2 + (n/2)^2 = n^2/2$.
- Se riunendo i due sottovettori ordinati si potesse riottenere il vettore originario tutto ordinato avremmo dimezzato il tempo di calcolo
- Se questo ragionamento si potesse applicare anche immaginando una decomposizione del vettore originario in quattro, si avrebbe un tempo di calcolo totale pari a: $(n/4)^2 + (n/4)^2 + (n/4)^2 + (n/4)^2 = n^2/4$.
- Se si potesse dividere in 8, si avrebbe un tempo ancora più basso, e così via dicendo.

Quick Sort

- Il ragionamento descritto prima non funziona sempre.



- **Il vincolo da porre è chiaro:** possiamo applicare questo metodo solo se il massimo elemento in A1 è inferiore o uguale al minimo elemento di A2 (nel caso di **ordinamento crescente**)



Quick Sort: Come funziona

- Nell'ambito del vettore viene scelto un elemento «a caso», denominato pivot
- Vengono eseguiti confronti e scambi in modo da disporre da una parte del vettore tutti gli elementi più piccoli del pivot e dall'altra parte tutti gli elementi più grandi del pivot
- L'algoritmo viene **ricorsivamente** applicato a ciascuna delle due parti (prima una e poi l'altra)
- La singola **ricorsione** termina quando ognuna delle due parti è ordinata
- La scelta del pivot è casuale: noi sceglieremo il pivot=l'elemento di indice m=(inf+sup)/2

Quick Sort: Come funziona

- Si consideri il seguente vettore, v, di n=10 elementi:

13	10	1	45	15	28	21	11	29	34
0	1	2	3	4	5	6	7	8	9

i=inf j=sup

- Scegliamo come pivot l'elemento di indice m=(inf+sup)/2, ovvero v[4]=15.
- L'indice i viene fatto incrementare fino a quando si trova un elemento maggiore o **uguale** al pivot
- L'indice j viene fatto decrementare fino a quando si trova un elemento minore o **uguale** al pivot

13	10	1	45	15	28	21	11	29	34
0	1	2	3	4	5	6	7	8	9

i j

Quick Sort: Come funziona

13	10	1	**45**	15	28	21	**11**	29	34
0	1	2	3	4	5	6	7	8	9
			i				**j**		

- Gli elementi di indice **i** e **j** vengono scambiati, e l'indice **i** viene incrementato, mentre **j** viene decrementato, ottenendo:

13	10	1	**11**	15	28	21	**45**	29	34
0	1	2	3	4	5	6	7	8	9
				i		**j**			

Quick Sort: Come funziona

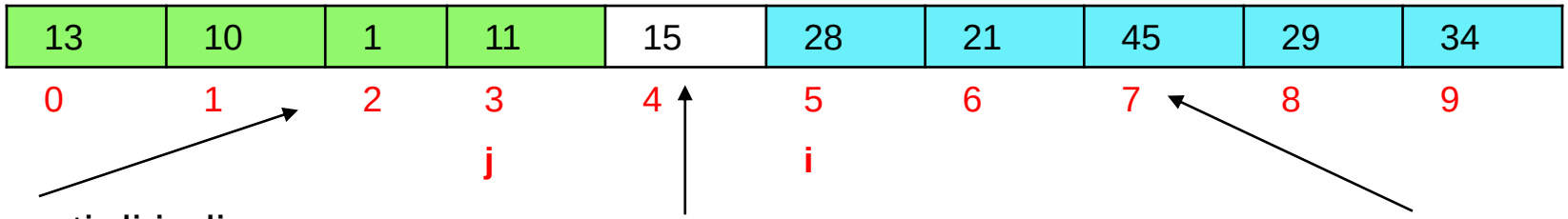
13	10	1	11	15	28	21	45	29	34
0	1	2	3	4	5	6	7	8	9
				i		**j**			

- L'indice **i** viene arrestato in quanto esso corrisponde al pivot. L'indice **j** viene fatto decrementare fino a quando esso perviene all'elemento 15, che è uguale al pivot. Gli indici sono dunque:

13	10	1	11	15	28	21	45	29	34
0	1	2	3	4	5	6	7	8	9
				i,j					

Quick Sort: Come funziona

- Gli elementi **i** e **j** non vengono scambiati, perché non avrebbe senso visto che **i** e **j** coincidono, e successivamente **i** viene incrementato e **j** viene decrementato, ottenendo l'inversione degli indici (**i** > **j**) e la conclusione del primo passo del QuickSort, Il vettore è così partizionato:



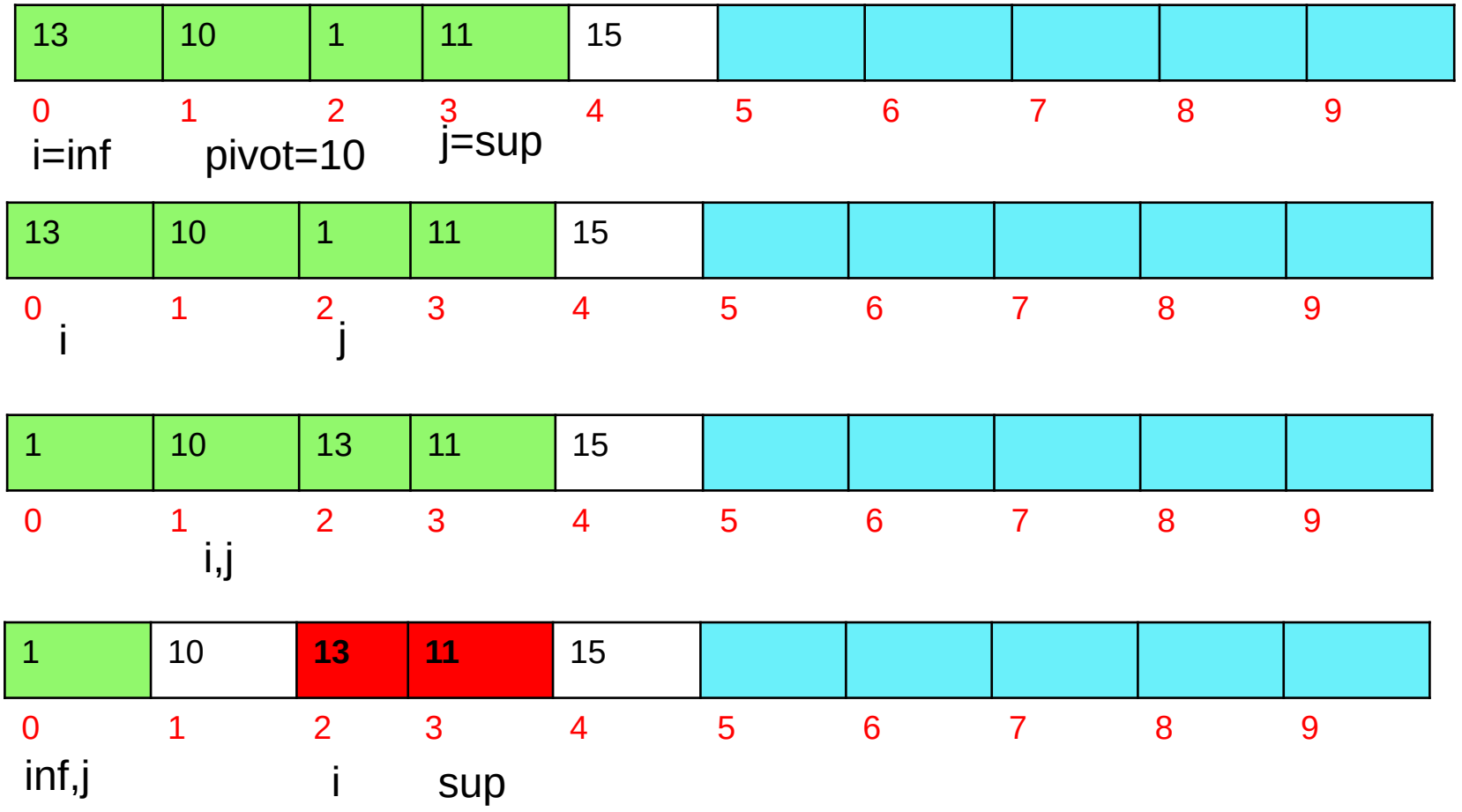
Gli elementi di indice appartenente a inf,..,j sono minori o uguali al pivot

Gli elementi di indice tra j+1,..,i-1 sono uguali al pivot (nel nostro esempio c'è un solo elemento)

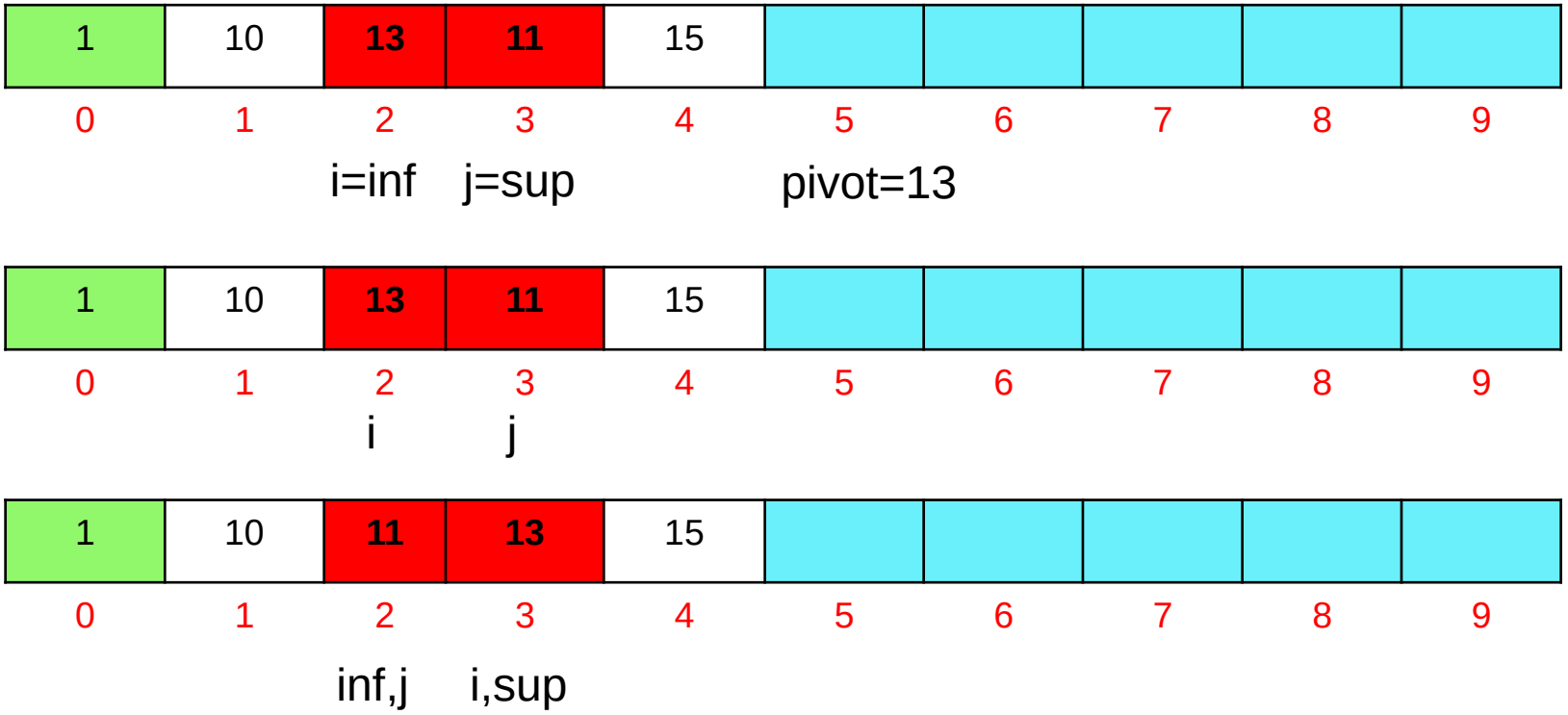
Gli elementi di indice appartenente a i ..sup sono superiori o uguali al pivot

- L'algoritmo procede ricorsivamente operando sui vettori delimitati dagli indici (inf,..,j) e (i,..,sup).

Quick Sort: Come funziona



Quick Sort: Come funziona



- Fine ricorsione sul sotto-vettore di sinistra, si procede su quello di destra

Quick Sort: Come funziona

- Perché: L'indice i viene fatto incrementare fino a quando si trova un elemento maggiore o **uguale** al pivot ?

13	10	1	15	90	28	21	45	29	34
0	1	2	3	4	5	6	7	8	9
i				pivot=90					j

- L'incremento dell'indice i non si arresterebbe mai, se non ci fosse la condizione "**uguale** al pivot"

Quick Sort: Come funziona

- Perché: L'indice j viene fatto decrementare fino a quando si trova un elemento minore o **uguale** al pivot ?

13	10	1	15	0	28	21	45	29	34
0	1	2	3	4	5	6	7	8	9
i				pivot=0					j

- Il decremento dell'indice j non si arresterebbe mai, se non ci fosse la condizione "**uguale** al pivot"

Codifica C del Quick Sort

```
1.  void QSort (tipobase v[], long inf, long sup)
2.  {
3.      tipobase pivot=v[(inf+sup)/2];
4.      long i=inf, j=sup;
5.      while (i<=j)  {
6.          while (v[i]<pivot) i++;
7.          while (v[j]>pivot) j--;
8.          if (i<j) scambia(v,i,j);
9.          if (i<=j) {i++; j--;}
10.     }
11.     if (inf<j) QSort(v,inf,j);
12.     if (i<sup) QSort(v,i,sup);
13. }
```

```
1.  void scambia (tipobase v[], long i, long j)
2.  {
3.      tipobase tmp=v[i];
4.      v[i]=v[j];
5.      v[j]=tmp;
6.  }
```

Prestazioni del Quick Sort

- Le prestazioni del Quick Sort dipendono dalla scelta del pivot, ma si dimostra che **mediamente** le sue prestazioni sono n*logn

- Ad ogni ricorsione si possono verificare diverse condizioni, che variano da un caso peggiore ad un caso migliore
  - **Caso migliore**
  - **Caso peggiore**

Quick Sort: caso migliore

13	10	1	45	15	12	21	15	29	34
0	1	2	3	4	5	6	7	8	9
i									j

- Scegliamo come pivot l'elemento di indice m = (inf+sup)/2, ovvero v[4]=15; casualmente questo coincide con l'elemento mediano m del vettore.
- *L'elemento mediano è quello tale che il numero di elementi del vettore più grandi di lui è circa uguale al numero di elementi del vettore che più piccoli di lui.*
- Il numero di elementi più piccoli di 15 è 4, mentre il numero di elementi più grandi di 15 è 4.

Quick Sort: caso migliore (cont.)

- L'indice **i** viene incrementato fino a quando non viene trovato un elemento più grande o uguale al pivot. Nel nostro esempio l'indice **i** si arresta in corrispondenza dell'elemento 45. Per quanto riguarda l'indice **j** esso viene spostato fino a quando non si perviene all'elemento 15, uguale al pivot.

13	10	1	45	15	12	21	15	29	34
0	1	2	3	4	5	6	7	8	9
			i				j		

- Gli elementi di indice **i** e **j** vengono scambiati, e l'indice **i** viene incrementato, mentre **j** viene decrementato, ottenendo:

13	10	1	15	15	12	21	45	29	34
0	1	2	3	4	5	6	7	8	9
				i		j			

Quick Sort: caso migliore (cont.)

- L'indice **i** viene arrestato in quanto esso corrisponde al pivot. L'indice **j** viene fatto decrementare fino a quando esso perviene all'elemento 12, che è inferiore al pivot:

13	10	1	15	15	12	21	45	29	34
0	1	2	3	4	5	6	7	8	9
				i	j				

- Gli elementi **i** e **j** vengono scambiati e successivamente **i** viene incrementato e **j** viene decrementato, ottenendo:

13	10	1	15	12	15	21	45	29	34
0	1	2	3	4	5	6	7	8	9
				j	i				

- La prima passata dell'algoritmo QuickSort si conclude, perché i due indici **i** e **j** si sono invertiti.

Quick Sort: caso migliore (cont.)

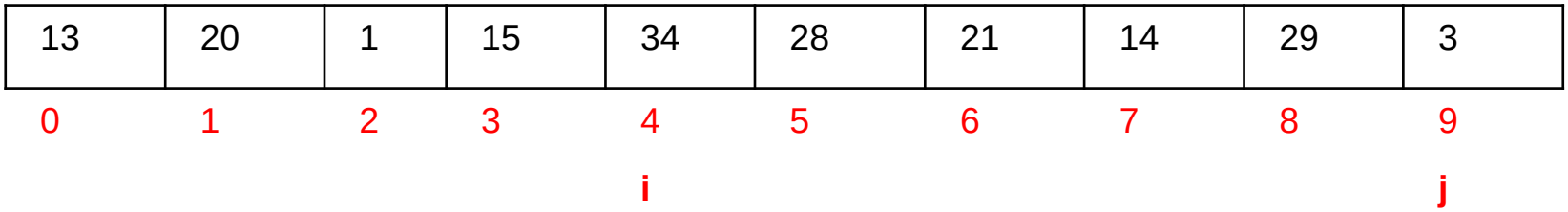
- Come si vede alla fine della prima passata di ordinamento risulta che:
  - tutti gli elementi di indice appartenente a inf,..,j sono minori o uguali del pivot
  - tutti gli elementi di indice appartenente a i,..,sup sono maggiori o uguali del pivot
  - non ci sono elementi di indice appartenente a j+1,..,i-1.
- Come si vede l'esempio considerato rappresenta il caso migliore perché il vettore originario è stato decomposto in due vettori che hanno entrambi dimensione uguale e pari a metà della dimensione iniziale.
- L'algoritmo procede ricorsivamente operando sui vettori delimitati dagli indici (inf,..,j) e (i,..,sup).

Quick Sort: caso peggiore

13	20	1	15	34	28	21	14	29	3
0	1	2	3	4	5	6	7	8	9
i									j

- Se scegliamo come pivot l'elemento di indice m=(inf+sup)/2, ovvero v[4]=34, questo casualmente coincide con l'elemento maggiore del vettore.

Quick Sort: caso peggiore (cont)



- L'indice **i** viene incrementato fino a quando non viene trovato un elemento più grande o uguale al pivot. Nel nostro esempio l'indice **i** si arresta in corrispondenza del pivot.
- L'esempio mette in evidenza il motivo di incrementare l'indice **i** fino a quando si trova un elemento più grande o uguale al pivot. Se non ci fosse la condizione uguale, nel nostro esempio l'indice **i** verrebbe continuamente incrementato oltre la dimensione del vettore.
- Per quanto riguarda l'indice **j** esso non viene spostato in quanto l'elemento **j**-esimo è inferiore al pivot.

Quick Sort: caso peggiore (cont)

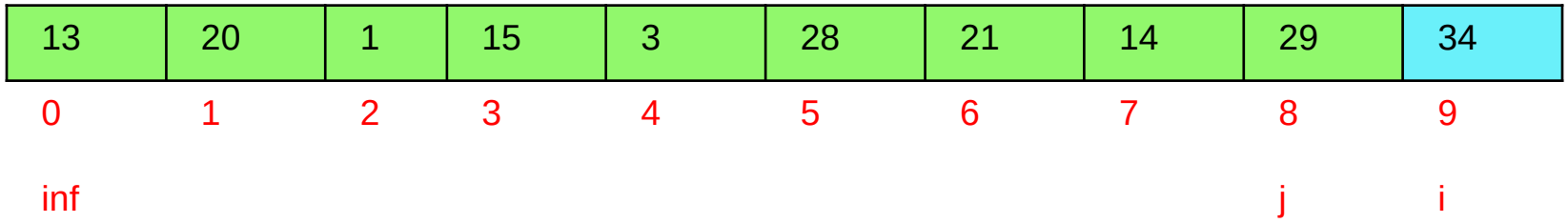
- Gli elementi di indice **i** e **j** vengono scambiati, e l'indice **i** viene incrementato, mentre **j** viene decrementato, ottenendo:

13	20	1	15	3	28	21	14	29	34
0	1	2	3	4	5	6	7	8	9
					i			j	

- L'indice **i** viene quindi fatto incrementare fino a quando arriva all'elemento 34, che è pari al pivot. L'indice **j** non viene fatto decrementare perché si riferisce ad un elemento già inferiore al pivot.

13	20	1	15	3	28	21	14	29	34
0	1	2	3	4	5	6	7	8	9
								j	i

Quick Sort: caso peggiore (cont)



- Siccome **i** > **j**, la prima ricorsione è finita:
  - tutti gli elementi di indice appartenente a inf,..,j sono minori o uguali del pivot. **Il numero di tali elementi è n-1**
  - vi è un solo elemento di indice i,..,sup (uguale al pivot)
  - non ci sono elementi di indice appartenente a j+1,..,i-1.
- L'algoritmo procede quindi ricorsivamente operando SOLO sul vettore delimitati dagli indici (inf,..,j)
- E' chiaro che nella successiva ricorsione, le cose potrebbero cambiare, ma si capisce come la scelta del pivot influenza le prestazioni del QuickSort

Quick Sort: calcolo tempo esecuzione

```
#include<time.h>

#define N 500

int vettore[N];

int main(void)
{
   time_t tstart,tend;

   printf("\nPremi un tasto per iniziare ad ordinare ");
   getchar();
   tstart = time(NULL);
   QuickSort(vettore,0,N-1);
   tend = time(NULL);
   printf("\nHo finito, ho impiegato %f secondi ",difftime(tend,tstart));
 }
```