Ordinamento e Ricerca

Ricerca Sequenziale

Necessaria quando gli elementi non sono ordinati.
Confronta la chiave mediamente con la metà degli elementi presenti.

```
int ricercaLineare (int vettore[], int dim, int chiave)
{
   for (int i = 0; i < dim; i++)
      if (vettore[i] == chiave) return i;
   return -1;
}
```

Ricerca Binaria

Se l’array è ordinato possiamo definire un algoritmo più efficiente?

Ricerca Binaria

Se l'array è ordinato possiamo definire un algoritmo più efficiente?

Inizio dal centro, successivamente mi sposto nel sottoarray destro o sinistro, dimezzando ogni volta il numero di elementi da confrontare.

Ricerca Binaria

Inizio dal centro, successivamente mi sposto nel sottoarray destro o sinistro, dimezzando ogni volta il numero di elementi da confrontare.

```
int RicercaBinaria (TipoDato v[], int basso, int alto, Tipodato chiave)
{
   int centrale;
   TipoDato valorecentrale;
   while (basso <= alto)
   {
      centrale = (basso + alto)/2;     // indice elemento centrale
      valoreCentrale = v[centrale];    // valore dell'indice centrale
      if (chiave == valoreCentrale)
         return centrale; // trovato valore,restituisce posizione
      else if (chiave < valoreCentrale)
         alto = centrale - 1;       // andare al semivettore basso
      else basso = centrale + 1;     // andare al semivettore alto
   }
   return -1; // elemento non trovato
}
```

Complessità della Ricerca

	Caso Migliore	Caso Medio	Caso Peggiore
Sequenziale			
Binaria			

Complessità della Ricerca

	Caso Migliore	Caso Medio	Caso Peggiore
Sequenziale	O(1)	O(n)	O(n)
Binaria	O(1)	O(logn)	O(logn)

Ordinamento di sequenze

- Insertion sort
- Selection sort
- Bubblesort
- Mergesort
- Quicksort

Insertion Sort

L’**ordinamento per inserimento** diretto consiste nell’inserire un elemento alla volta nella posizione che gli spetta in un vettore già ordinato, partendo da un vettore che contiene un solo elemento.

Si aggiungono gli altri elementi uno per volta, posizionandoli direttamente nella posizione corretta.

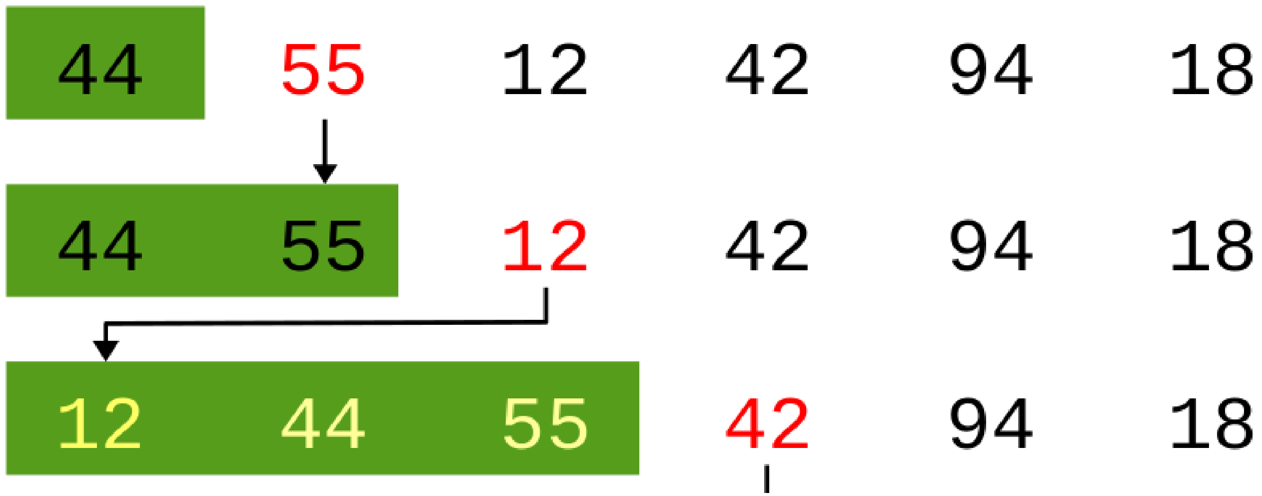
Insertion Sort

44	55	12	42	94	18

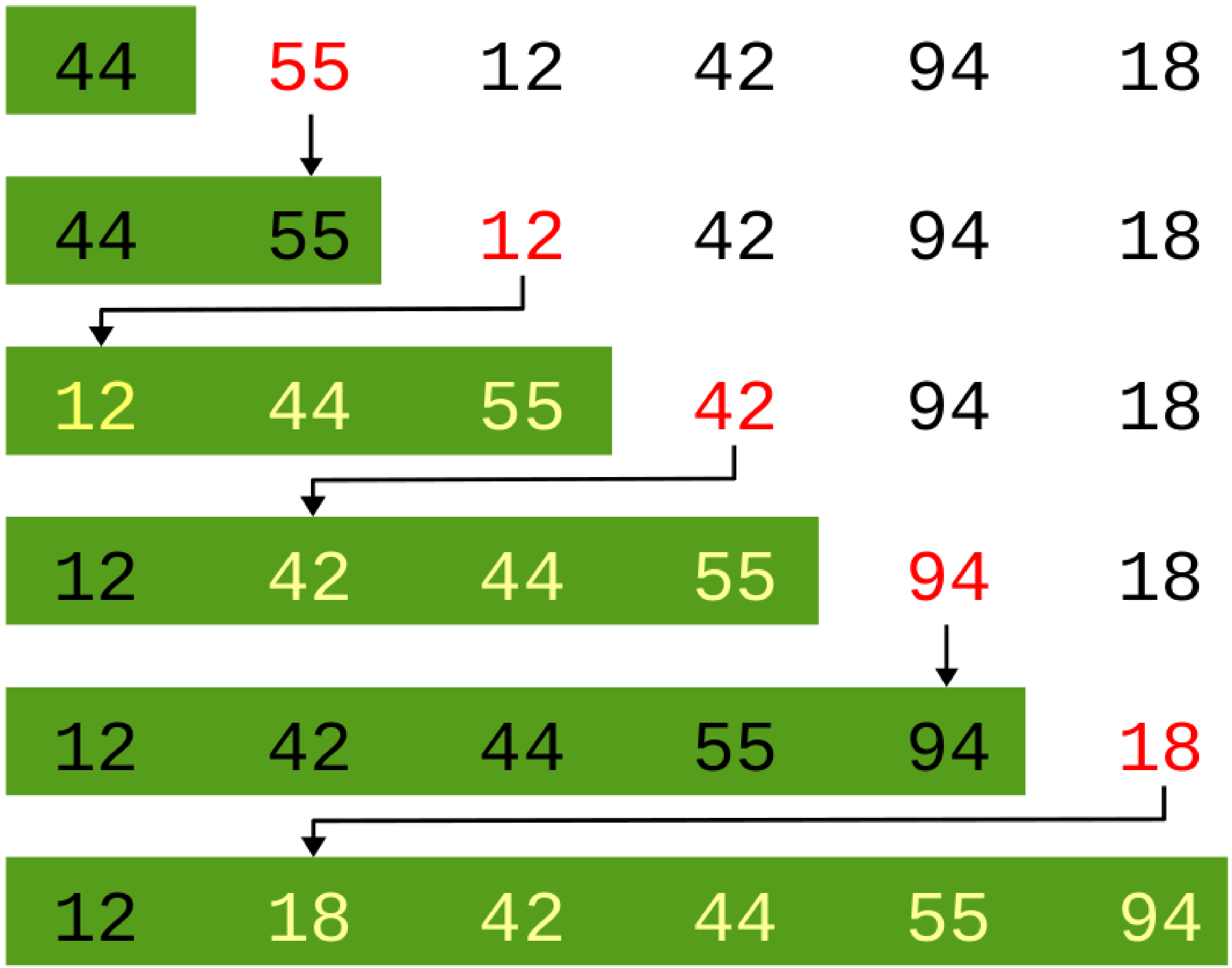
Insertion Sort

44	55	12	42	94	18
44	55	12	42	94	18

Insertion Sort



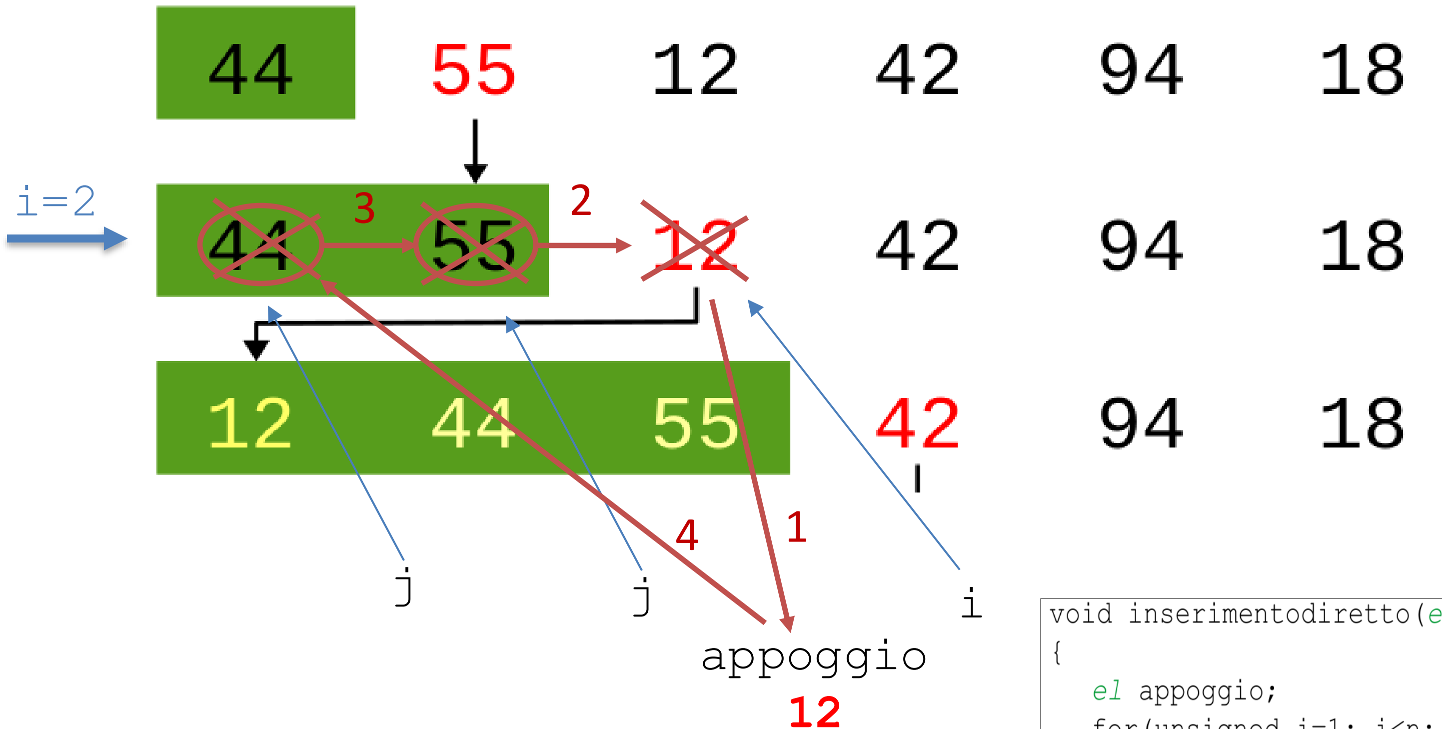
Insertion Sort



Insertion Sort

```
void inserimentodiretto(el v[], unsigned n)
{
   el appoggio;
   for(unsigned i=1; i<n; i++) {
      appoggio = v[i];
      int j = i-1;
      while((j >= 0) && (v[j] > appoggio)) {
         v[j + 1] = v[j];
         j--;
      }
      v[j+1] = appoggio;
   }
}
```

Insertion Sort




```
void inserimentodiretto(el v[], unsigned n)
{
   el appoggio;
   for(unsigned i=1; i<n; i++) {
      appoggio = v[i];
      int j = i-1;
      while((j >= 0) && (v[j] > appoggio)) {
         v[j + 1] = v[j];
         j--;
      }
      v[j+1] = appoggio;
   }
}
```

Insertion Sort

	Caso Migliore	Caso Medio	Caso Peggiore
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$

Possiamo migliorare l'insertion sort utilizzando la ricerca binaria per collocare correttamente l'elemento a[i] nel sottoarray ordinato.

Questo riduce il numero di passi per trovare la posizione corretta da $O(n^2)$ ad $O(n \log n)$, tuttavia, ognuno degli elementi deve essere spostato di una posizione e ciò richiede comunque un costo totale di $O(n^2)$ nel caso medio e peggiore.

Selection Sort

L’ordinamento per selezione diretta seleziona l’elemento minore del sottoarray non ordinato e lo posiziona al primo posto.

Selection Sort

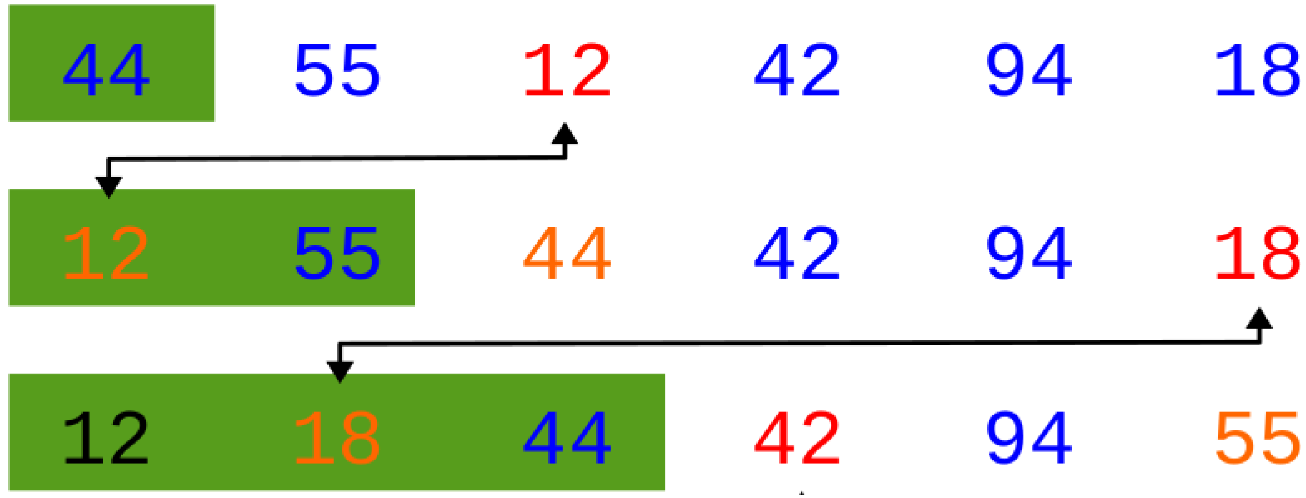
44 55 12 42 94 18

Selection Sort

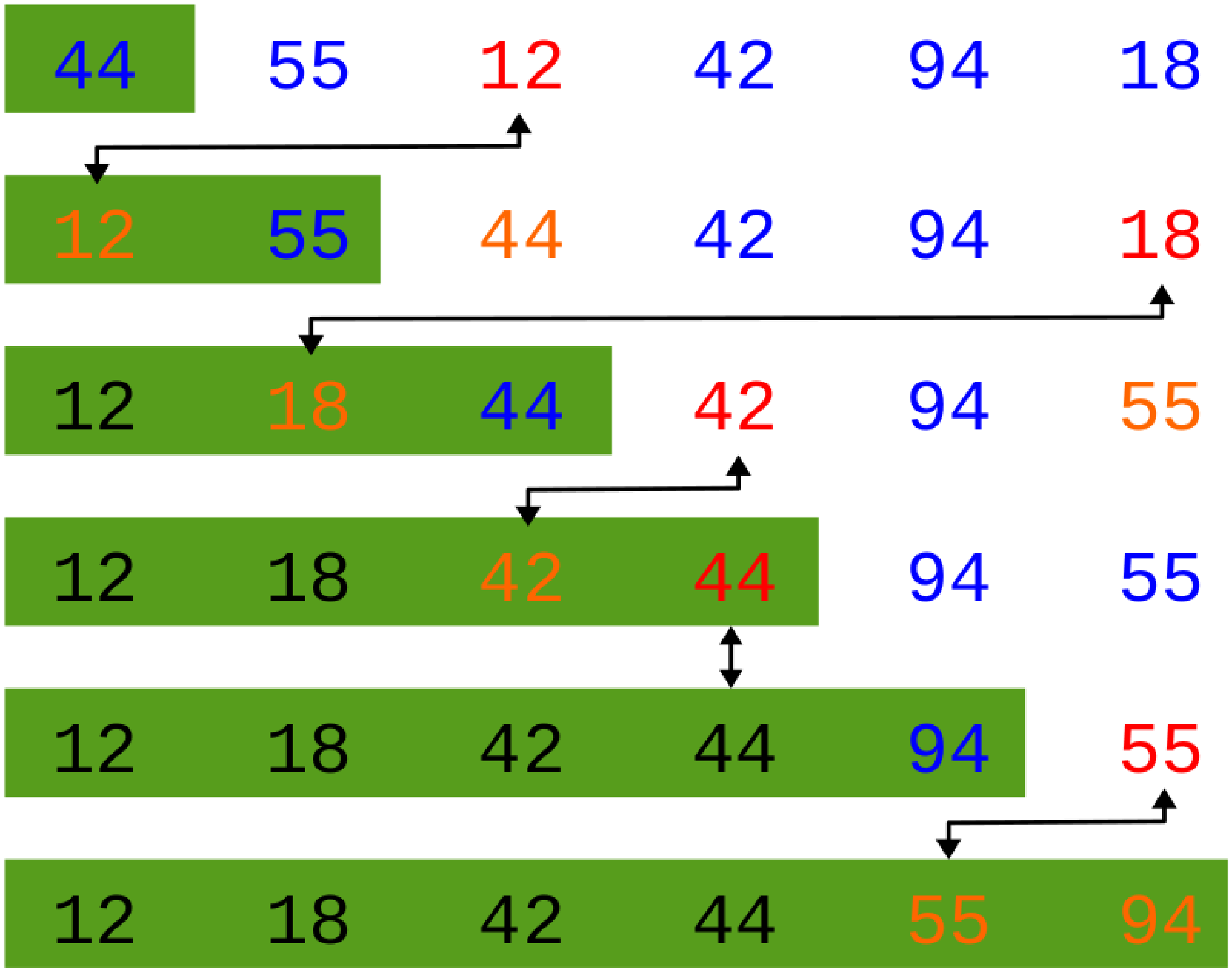
44 55 12 42 94 18

12 55 44 42 94 18

Selection Sort



Selection Sort



Selection Sort

```
void selezionediretta(el v[], unsigned n)
{
   el appoggio;
   unsigned posminimo;
   for(unsigned i=0; i<n-1; i++) {
      posminimo = i;
      unsigned j = i+1;
      while(j < n) {
         if (v[j] < v[posminimo]) posminimo = j;
         j++;
      }
      appoggio = v[i];
      v[i] = v[posminimo];
      v[posminimo] = appoggio;
   }
}
```

Costo $O(n^2)$ in qualsiasi caso

Esercizi

1. Implementare l’algoritmo Insertion Sort
2. Migliorare l’algoritmo precedente sfruttando la ricerca binaria
3. Implementare l’algoritmo Selection Sort
4. Applicare la ricerca sequenziale per contare il numero di volte che un elemento compare in un array.

Bubble Sort

Per ogni iterazione si confrontano gli elementi adiacenti e si scambiano i loro valori quando il primo è maggiore del secondo. Come conseguenza, abbiamo che il maggiore ‘’risale’’ fino alla cima del vettore ad ogni iterazione.

Dopo *n-1* iterazioni l’array risulterà ordinato.

Bubble Sort

44	55	12	42	94	18

Bubble Sort

44	55	12	42	94	18
44	12	55	42	94	18

Bubble Sort

44	55	12	42	94	18
44	12	55	42	94	18
44	12	42	55	94	18

Bubble Sort

44	55	12	42	94	18	
44	12	55	42	94	18	
44	12	42	55	94	18	
44	12	42	55	18	94	← *Il massimo è risalito*
12	44	42	55	18	94	
12	42	44	55	18	94	
12	42	44	18	55	94	
12	42	18	44	55	94	
12	18	42	44	55	94	

Bubble Sort

```
Ripeti n-1 volte {
        ripeti n-1 volte {
                confronta elementi adiacenti,
                se è il caso effettua uno swap
        }
}
```

Possiamo migliorare questa strategia ?

Bubble Sort

```
Ripeti n-1 volte {
        ripeti n-1 volte {
                confronta elementi adiacenti,
                se è il caso effettua uno swap
        }
}
```

Possiamo migliorare questa strategia ?

```
Ripeti fino a quando ci sono scambi da fare{
        ripeti n-1 volte {
                confronta elementi adiacenti,
                se è il caso effettua uno swap
        }
}
```

Bubble Sort

```
void bubblesort(el v[], unsigned n)
{
    el appoggio;
    bool scambio;
    scambio = true;
    while (scambio) {
        scambio = false;
        for(unsigned i=0; i<n-2; i++) {
            if (v[i] > v[i+1]) {
                appoggio = v[i];
                v[i] = v[i+1];
                v[i+1] = appoggio;
                scambio = true;
            }
        }
    }
}
```

Bubble Sort

	Caso Migliore	Caso Medio	Caso Peggiore
BubbleSort	$O(n^2)$	$O(n^2)$	$O(n^2)$
BubbleSort + flag	$O(n)$	$O(n^2)$	$O(n^2)$

Merge Sort

Questo algoritmo implementa il paradigma *divide et impera:*

- L'input di dimensione *n* viene partizionato in due parti di lunghezza *n/2*.

- Le due sottosequenze vengono ordinate in maniera ricorsiva fino a quando si ottengono delle sequenze composte da un solo elemento.

- A questo punto la procedura *merge* unisce due sottosequenze ordinate.

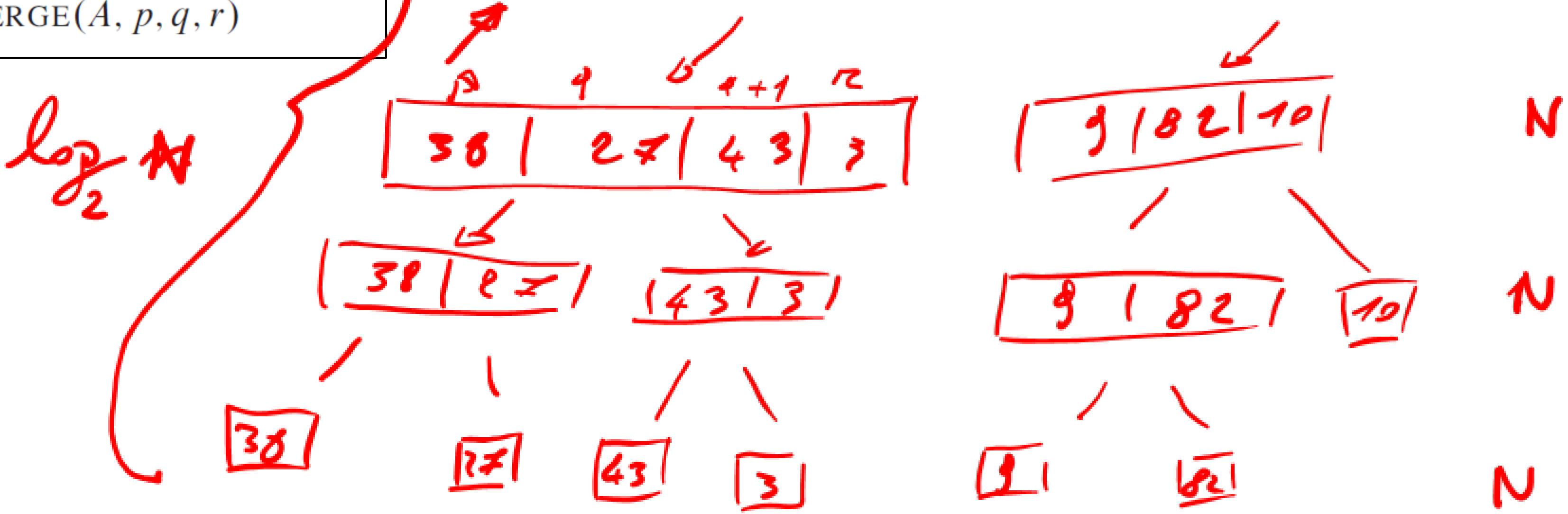
```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Merge Sort

MERGE-SORT(A, p, r)

```
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

p			q	q+1		r
38	27	43	3	9	82	10



$O(N \log N)$

Merge Sort

	Caso Migliore	Caso Medio	Caso Peggiore
MergeSort	O(nLogn)	O(nLogn)	O(nLogn)

La complessità è la medesima in tutti e tre i casi perché l'algoritmo divide sempre le sequenze a metà impiegando un tempo O(log n) e le unisce impiegando un tempo lineare.

Quick Sort

Il Quicksort è l'algoritmo di ordinamento più efficiente. Si basa sulla divisione del vettore in tre partizioni:

- Centrale: contenente un solo elemento detto *pivot*
- Sinistra: contenente tutti gli elementi minori del *pivot*
- Destra: contenente tutti gli elementi maggiori del *pivot*

Come conseguenza avremo che tutti gli elementi della partizione sinistra saranno minori del più piccolo della partizione di destra.

Si applica ricorsivamente l'algoritmo sulle partizioni sinistra e destra fino ad ordinare tutto il vettore.

Il *pivot* può essere scelto a caso.

Quick Sort

44	12	55	42	94	18

Quick Sort

Possiamo definire una procedura *partition* che si occupa di effettuare la partizione e restituire la posizione del pivot.

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

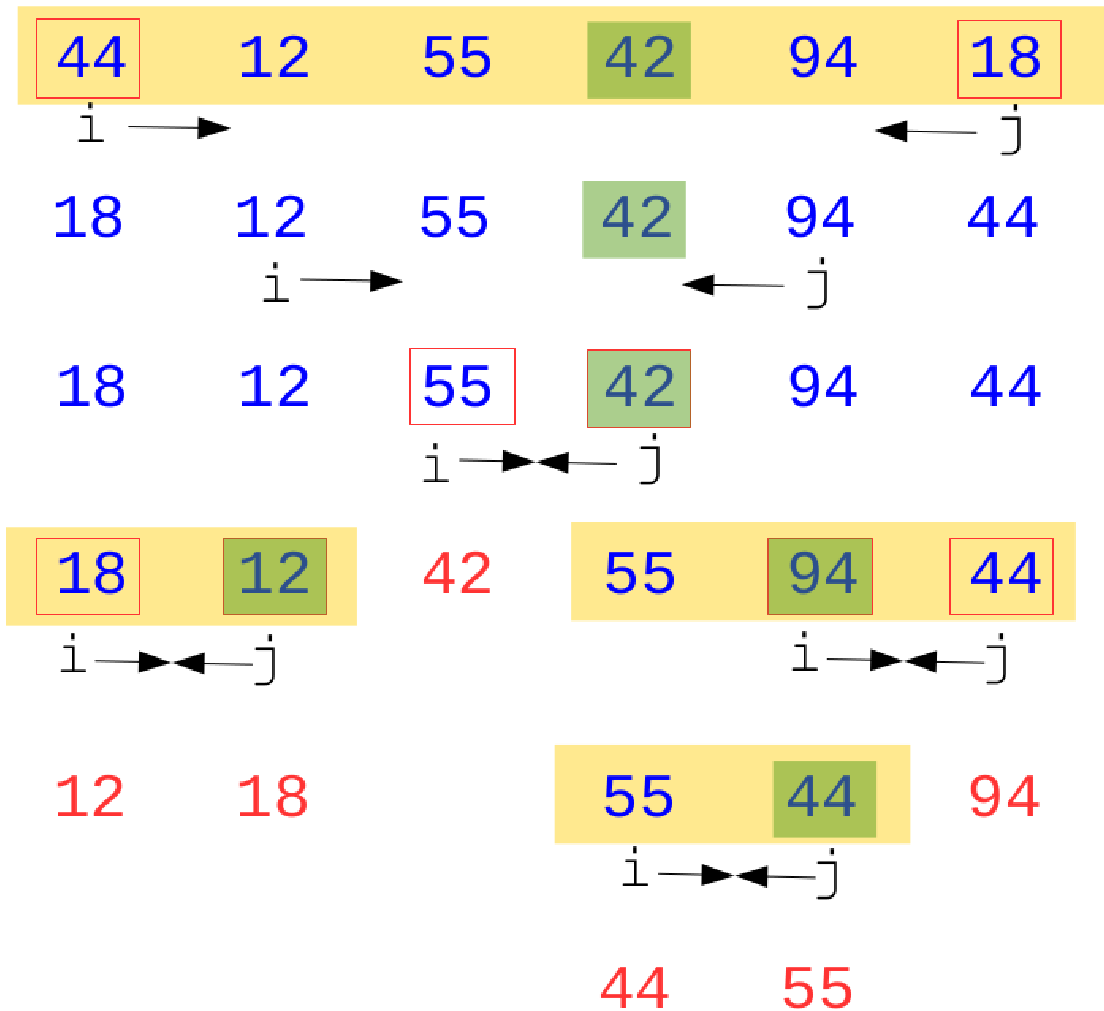
```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

Quick Sort

Implementazione alternativa senza la procedura partition.

```
void QuickSort(el* v, int n) {
   QuickSort(v, 0, n-1);}
void QuickSort(el v[], int s, int d)
{
    int i = s, j = d;
    el tmp;
    el pivot = v[(s + d) / 2];
    while (i <= j) {                  // PARTIZIONE
        while (v[i] < pivot) i++;
        while (v[j] > pivot) j--;
        if (i <= j) {
            tmp = v[i];
            v[i] = v[j];
            v[j] = tmp;
            i++;
            j--;
        }
    };
    if (s < j)          // RICORSIONE
    QuickSort(v, s, j);
    if (i < d)
    QuickSort(v, i, d);
}
```

Quick Sort



44 12 55 42 94 18
i j
18 12 55 42 94 44
i j
18 12 55 42 94 44
i j
18 12 42 55 94 44
i j i j
12 18 55 44 94
i j
44 55

Quick Sort

	Caso Migliore	Caso Medio	Caso Peggiore
QuickSort	O(nLogn)	O(nLogn)	O(n^2)

- **Caso peggiore:** quando le due partizioni sono formate da 0 ed n-1 elementi. In questo caso il partizionamento costa O(n) e se questo caso si verifica ad ogni chiamata ricorsiva avremo un costo totale di O(n^2).

- **Caso migliore:** bilanciamento massimo. Si verifica quando i due sottoproblemi hanno dimensione circa n/2. In questo caso il costo è O(nLogn)

- **Caso medio:** è possibile dimostrare che anche con una ripartizione sproporzionata ad ogni livello di ricorsione, il quicksort viene eseguito nel tempo O(nLogn). Questo perché qualsiasi ripartizione con proporzionalità costante produce una ricorsione di profondità O(logn) il cui costo unitario è O(n).