ALGORITMI DI ORDINAMENTO

- ✓ Complessità del problema dell'ordinamento
- ✓ Naï ve sort
- ✓ Bubble sort
- ✓ Insertion sort
- ✓ Quick sort
- ✓ Merge sort

COMPLESSITÀ DEL PROBLEMA DELL'ORDINAMENTO

- Dato un insieme A di n elementi distinti $\{v_1,\ldots,v_n\}$ esistono n! permutazioni
- Risolvere il **problema dell'ordinamento** significa trovare l'unica permutazione ordinata ovvero

$$\forall\, v_i, v_j: \quad i < j \quad \text{si ha} \quad v_i < v_j$$

Il problema dell'ordinamento ha una delimitazione inferiore $\Omega(f(n))$?

Dati v_i e v_j con i < j, è possibile partizionare l'insieme delle permutazioni in due sottoinsiemi:

- le permutazioni con $v_i < v_j$ (ammissibile)
- le permutazioni con $v_i > v_j$ (non ammissibile)

Quante partizioni sono necessarie per giungere ad un sottoinsieme di una sola permutazione ?

È sufficiente trovare **k** tale per cui $\mathbf{2^k = n!}$

Data la formula di Stirling, $n! \quad (n/e)^n \quad (2 \pi n)$:

$$2^k \quad (n/e)^n \quad (2 \pi n)$$

$$\log_2 2^k = \log_2 (n/e)^n \quad (2 \pi n)$$

$$k = n \log_2(n/e) + \tfrac{1}{2}\log_2 (2 \pi n)$$

ovvero $\Omega(n \log_2 n)$

ALGORITMI DI ORDINAMENTO

Un algoritmo di ordinamento ha come scopo quello di *ordinare una sequenza di elementi* di un determinato tipo rispettando una certa *relazione d'ordine*.

In un linguaggio di programmazione, tali elementi sono di norma memorizzati in un vettore.

Algoritmi di ordinamento più importanti:

- ***naï ve sort*** (semplice, intuitivo, poco efficiente)
- ***bubble sort*** (semplice, un po' più efficiente)
- ***shell sort*** (generalizza il bubbole sort, abbastanza efficiente)
- ***insert sort*** (intuitivo, abbastanza efficiente)
- ***quick sort*** (non intuitivo, alquanto efficiente)
- ***merge sort*** (non intuitivo, molto efficiente)

Ogni algoritmo riceve in ingresso:

- il vettore v da ordinare del tipo specificato
- la dimensione n del vettore

NAÏVE SORT o SELECTION SORT

SPECIFICA

```
while (n>1){
    <trova la posizione p del massimo>
    if (p<n-1) <scambia v[n-1] e v[p]>
    <considera come nuovo vettore v' il
     vettore v escluso l'ultimo elemento>}
```

CODIFICA

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

int trovaPosMax(int v[], int n){
int i, p=0;    /* ipotesi: max = v[0] */
for (i=1; i<n; i++)
    if (v[p]<v[i]) p=i;
return p;}
```

VALUTAZIONE COMPLESSITÀ

Nel caso peggiore l'algoritmo richiede un numero di confronti e di scambi pari a $\sum_{i=0}^{n-1} i = \frac{n*(n-1)}{2}$. La complessità asintotica è quindi **O(n^2)**.

BUBBLE SORT

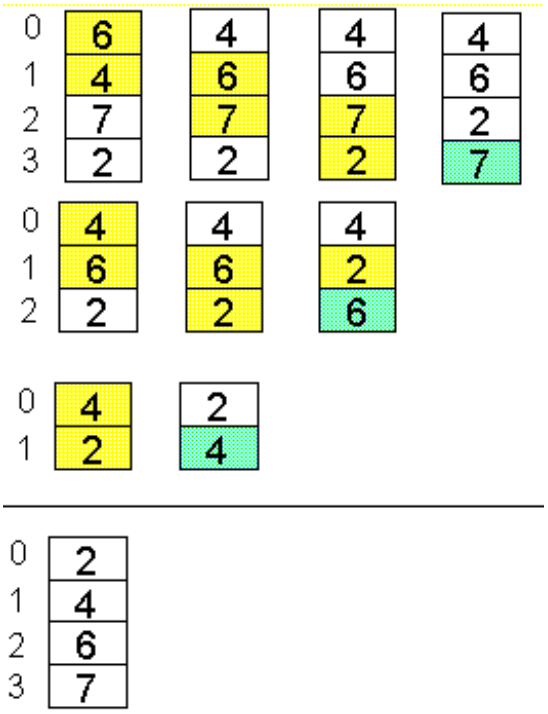
- È meno intuitivo, ma egualmente semplice.
- Corregge il difetto principale del naï ve sort ovvero quello di non accorgersi quando il vettore è giàordinato.
- Opera tramite “passate” successive sul vettore.
- A ogni “passata”, considera una ad una tutte le possibili coppie di elementi adiacenti, scambiandoli se sono nell’ordine errato.
  Al termine della passata, quindi, l’elemento massimo saràin fondo alla parte di vettore considerato.
- Se non si verificano scambi, il vettore è già ordinato e quindi l’algoritmo termina.

CODIFICA

```
void bubbleSort(int v[], int dim){
int i;
bool ordinato = false;
while (dim>1 && !ordinato) {
   ordinato = true;    /* hp: è ordinato */
   for (i=0; i<dim-1; i++)
      if (v[i]>v[i+1]) {
         scambia(&v[i],&v[i+1]);
         ordinato = false;}
   dim--;}}
```

BUBBLE SORT (segue)

ESEMPIO



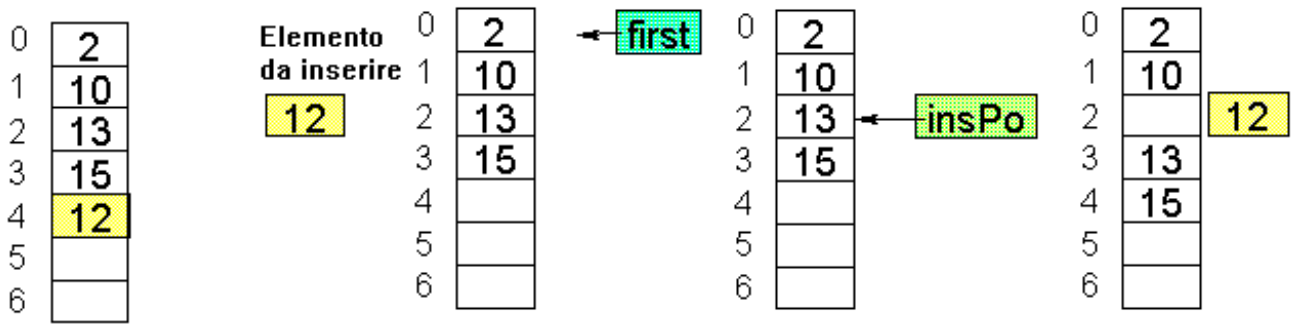
VALUTAZIONE COMPLESSITÀ

- La complessità dell'algoritmo dipende dalla configurazione dell'input
- Nel *caso peggiore* l'algoritmo richiede un numero di confronti analogo al precedente. La complessità asintotica è **$O(n^2)$**
- Il *caso migliore* si verifica quando il vettore è già ordinato. Alla prima passata, con (n-1) confronti, l'algoritmo se ne accorge e non prosegue inutilmente.
- Il *numero di confronti* varia quindi fra $(n^2-n)/2$ e (n-1).

INSERT SORT

L'insert sort parte da un approccio diverso dai precedenti: per ottenere un vettore ordinato basta *costruirlo ordinato, inserendo ogni elemento al posto giusto*.

ESEMPIO



SCELTA DI PROGETTO

in un vettore di i elementi (numerati da 0 a i-1), l'elemento da inserire si trova nella posizione i-esima.

SPECIFICA

```
for (i=1; i<n; i++)
<inserisci nella posizione corretta del sotto-vettore ordinato v[0],…,v[i] l'elemento che si trova nella posizione
i-esima>
```

CODIFICA

```
void insertSort(int v[], int dim){
  int i;
  for (i=1; i<dim; i++) insMinore(v,i);}
```

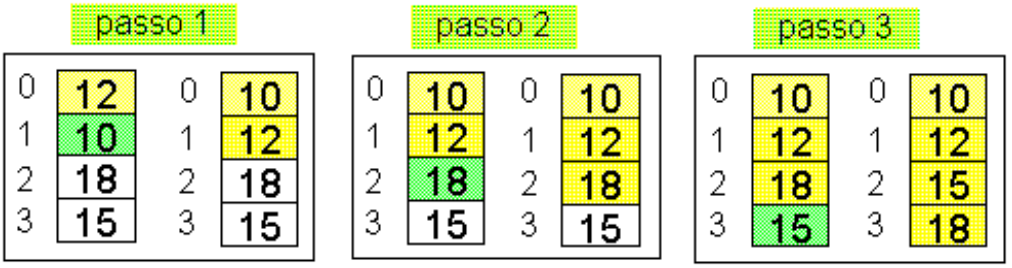
INSERT SORT (segue)

SPECIFICA

```
void insMinore(int v[], int lastpos){
<determina la posizione in cui inserire
l'elemento in posizione lastpos>
<crea lo spazio spostando gli altri el.>
<inserisci il nuovo elemento>}
```

CODIFICA

```
void insMinore(int v[], int lastpos){
int i, x = v[lastpos];
for (i = lastpos-1; i>=0 &&  x<v[i]; i--)
   v[i+1]= v[i];    /* crea lo spazio */
v[i+1]=x;}
```



VALUTAZIONE COMPLESSITÀ

- *Caso migliore* (vettore giàordinato). Bastano (n-1) confronti
- *Caso peggiore* (vettore ordinato al contrario). Il numero di confronti è $\sum_{i=1}^{n-1} i =$ (n*(n-1))/2: O(n^2).
- *Caso medio.* Ad ogni ciclo il nuovo elemento va inserito nella posizione centrale del vettore: $\sum_{i=0}^{n-1} (i/2) =$ (n*(n-1))/4 confronti.

QUICK SORT

- Il quick sort parte dall'idea che ordinare un vettore corto è molto meno costoso che ordinarne uno lungo.
- L'idea consiste nel *partizionare il vettore da ordinare in due sottovettori* delimitati da un elemento "sentinella" (***pivot***). L'obiettivo è avere nel primo sotto-vettore solo elementi *minori o uguali* al pivot, e nel secondo sotto-vettore solo elementi *maggiori* del pivot.
- I due sotto-vettori possono poi essere ordinati, separatamente, *riapplicando lo stesso procedimento* $\rightarrow$ **l'algoritmo è *ricorsivo*.**
- Il caso banale è costituito da vettori di un singolo elemento, che sono già ordinati.

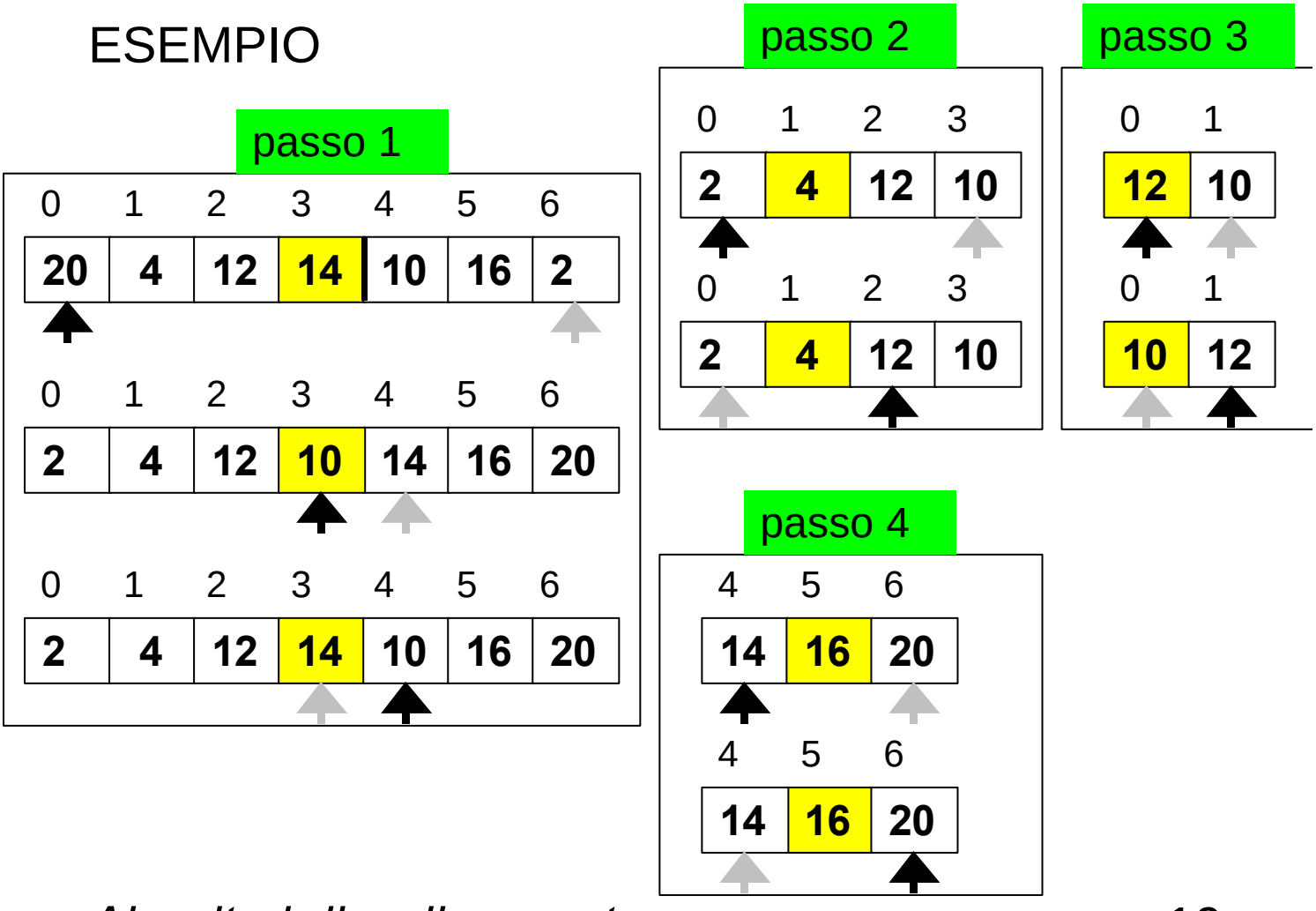
QUICK SORT (segue)

SPECIFICA

- scegliere un elemento come pivot
- partizionare il vettore nei due sottovettori:

  se il primo sotto-vettore contiene un elemento > pivot, e il secondo sotto-vettore contiene un elemento < pivot, i due elementi vengono scambiati.
- ordinare separatamente i due sottovettori (*ricorsione*: richiama quicksort sui due sottovettori)

ESEMPIO



QUICK SORT (segue)

SPECIFICA

```
void quickSort(int v[], int first, int last){
if (vettore non vuoto)
    <scegli come pivot l'elemento medio>
    <isola nella prima metà vettore gli
     elementi minori o uguali al pivot e
     nella seconda metà quelli maggiori>
    <richiama quicksort ricorsivamente
        sui due sottovettori >
}
```

CODIFICA

```
void quickSort(int v[], int first, int last){
int i, j, pivot;
if (first<last) {
   i = first;  j = last;
   pivot = v[(first + last)/2];
   do {
      while (v[i] < pivot) i++;
      while (v[j] > pivot) j--;
      if (i <= j) {
         scambia(&v[i], &v[j]);
         i++, j--;}
   } while (i <= j);
   quickSort(v, first, j);
   quickSort(v, i, last); }}
```

QUICK SORT (segue)

VALUTAZIONE COMPLESSITÀ

- La complessitàdell'algoritmo dipende dalla *scelta del pivot*:
- Se il pivot è scelto male, ovvero uno dei due sotto-vettori ha lunghezza zero, il numero di confronti è sempre $O(n^2)$.
- Se il pivot è scelto bene, in modo da produrre due sotto-vettori della stessa dimensione:
    - si hanno $\log_2 n$ attivazioni di quicksort
    - al passo k si opera su 2^k vettori di lunghezza $L = n/2^k$
- quindi, il numero di confronti ad ogni livello è n.

Il numero globale di confronti è **$O(n*\log_2 n)$**:

Quick sort è un algortimo ottimo per il problema dell'ordinamento

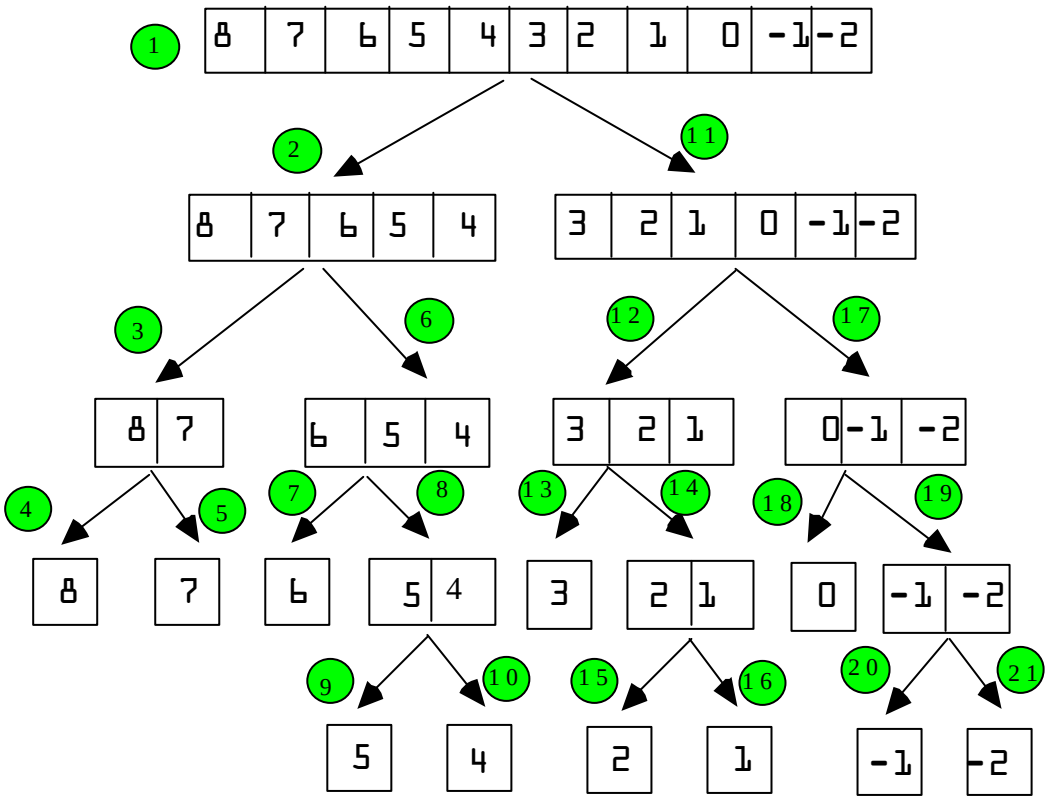
MERGE SORT

- E’ una variante del quick sort costruita in modo da avere *sempre* due sotto-vettori di ampiezza uguale. Il **merge sort garantisce sempre le prestazioni migliori possibili**, con complessità **O(n*log$_2$ n)**.

SPECIFICA

- spezza il vettore in due parti *della stessa dimensione*
- ordinali separatamente
- *fondi* i due sotto-vettori ordinati così ottenuti.

ESEMPIO



MERGE SORT (segue)

SPECIFICA

```
void mergeSort(int v[], int first, int last){
if (vettore v non vuoto)
   <partiziona il vettore in due metà>
   <richiama mergeSort ricorsivamente
       sui due sottovettori, se non vuoti>
   <fondi le due metà ordinate>}
```

CODIFICA

```
void mergeSort(int v[], int first, int last){
int mid;
if ( first < last ) {
   mid = (last + first) / 2;
   mergeSort(v, first, mid);
   mergeSort(v, mid+1, last);
   merge (v, first, mid+1, last);}}
void  merge(int v[], int i1, int i2, int last)
{int vout[N]; /* vettore temporaneo */
int i=i1, j=i2, k=i1;
while ( i <= i2-1 &&  j <= last ) {
   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 <= last) { vout[k] = v[j++]; k++;}
for (i=i1; i<=last; i++) v[i] = vout[i];}
```