ORDINAMENTO PER SELEZIONE

- Per l'analisi delle prestazioni di tale algoritmo di ordinamento, si considerano i due cicli `for` annidati: poiché i *confronti* avvengono nel ciclo interno si ha che

$$\sum_{i=0}^{n-2}(n-1-i) = n(n-1)/2 = O(n^2)$$

- Quindi una complessità quadratica. Per applicazioni pratiche con un elevato numero di dati è necessario trovare una soluzione migliore.

ORDINAMENTO PER SELEZIONE

- Solitamente si tiene conto solo dei confronti tra i *dati* e non tra *indici*, perchè il confronto tra indici è trascurabile se i dati sono grandi strutture.

- Risulta anche interessante valutare il numero di *scambi* effettuati da un algoritmo. L'ordinamento per selezione ha una complessità lineare sugli scambi, un risultato buono.

CONFRONTI

- Il confronto tra oggetti avviene attraverso i metodi dichiarati, ma non implementati, delle interfacce `Comparable` e `Comparator`.
- Per esempio:

```
Comparable obj1 = ...;
Object obj2 = ...;
```

`obj1.compareTo(obj2)`
- <0 se obj1 è minore di obj2
- =0 se obj1 è uguale a obj2
- >0 se obj1 è maggiore di obj2

ORDINAMENTO VELOCE

- Consideriamo un algoritmo molto diffuso: il *quicksort.* La versione di base fu inventata da Hoare nel 1960.

- È facilmente implementabile e richiede mediamente solo *n log(n)* operazioni.

- L'algoritmo di base presenta alcuni svantaggi, tuttavia è stato migliorato al punto da diventare il metodo ideale per un gran numero di applicazioni (una versione è usata nella classe `Arrays` di Java).

ORDINAMENTO VELOCE

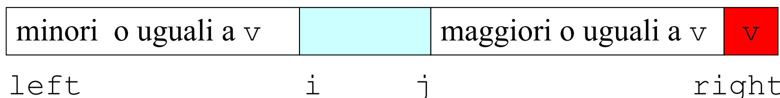
- I principali svantaggi dell'algoritmo di base sono i seguenti:
  - Nel caso peggiore ha una complessità quadratica.
  - Non è stabile. Un algoritmo di ordinamento è *stabile* se non altera l'ordine relativo di elementi dell'array aventi la stessa chiave.
- I principali vantaggi sono:
  - Una complessità media *O(n log n)*, che è ottima.
  - Opera sul *posto*: la dimensione delle "variabili ausiliarie" di cui ha bisogno è indipendente dalla dimensione dell'array da ordinare.

ORDINAMENTO VELOCE

- Il *quicksort* è un metodo che opera *partizionando* un array in *due parti* da ordinare indipendentemente.
- Lo scopo del partizionamento è quello di *riorganizzare* l'array in modo tale che: dato *l'elemento di partizionamento,* tutti gli elementi precedenti siano minori o uguali e quelli successivi siano maggiori o uguali di tale elemento di partizionamento.
- Poi si applica *ricorsivamente* ai due sotto-array lo stesso procedimento.

ORDINAMENTO VELOCE

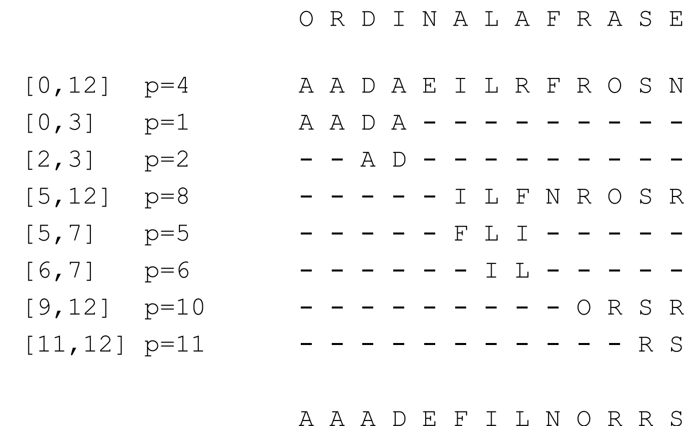
- La scelta dell'*elemento di partizionamento* avviene, in questa versione di base, in modo del tutto arbitrario (per es. elemento più a destra).
- Per *riorganizzare* l'array la strategia è quella di partire dai due estremi e scambiare gli elementi fuori posto.



- Dopo che gli indici di scansione `i` e `j` si sono incrociati, si deve porre l'*elemento di partizionamento* `v` nella posizione corretta, cioè scambiarlo con l'elemento più a sinistra della porzione a destra dell'array, e memorizzare tale indice `p` di partizionamento.

ORDINAMENTO VELOCE

- Vediamo un esempio con un array di lettere:



ORDINAMENTO VELOCE

```
public static void quicksort(Object[]
            data, int left, int right){
   if (left>=right) return;

   int p = partition(data,left,right);

   quicksort(data,left,p-1);
   quicksort(data,p+1,right);
}
```


ORDINAMENTO VELOCE

```
static int partition(Object[] data, int left, int
                                              right) {
  int i=left-1, j= right;
  Comparable v = (Comparable) data[right];
  for(;;){
     while(((Comparable)data[++i]).compareTo(v)<0);
     while(v.compareTo(data[--j])<0)
           if (j==left) break;
     if (i>=j) break;
     swap(data,i,j);
  }
  swap(data,i,right);
  return i;
}
```

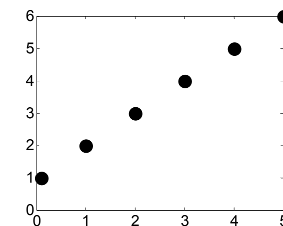

ORDINAMENTO VELOCE

- Notare che la ricorsione avviene sempre su array con dimensioni strettamente inferiori a quelle di partenza.
- L’efficienza dell’ordinamento dipende da quanto è bilanciato il partizionamento e quindi dal valore dell’elemento di partizionamento.
- Nel caso medio ha una complessità $O(n \log n)$, nel caso peggiore (array ordinato) $O(n^2)$.

CARATTERISTICHE DINAMICHE

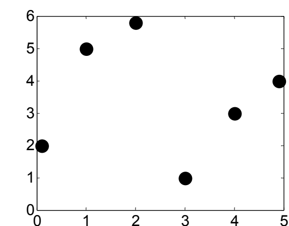
- Visualizziamo graficamente le caratteristiche dinamiche degli algoritmi di ordinamento.
- Il grafico rappresenta i valori dell’array `v` in funzione dell’indice:

caso ordinato



`int[] v={1,2,3,4,5,6};`

non ordinato

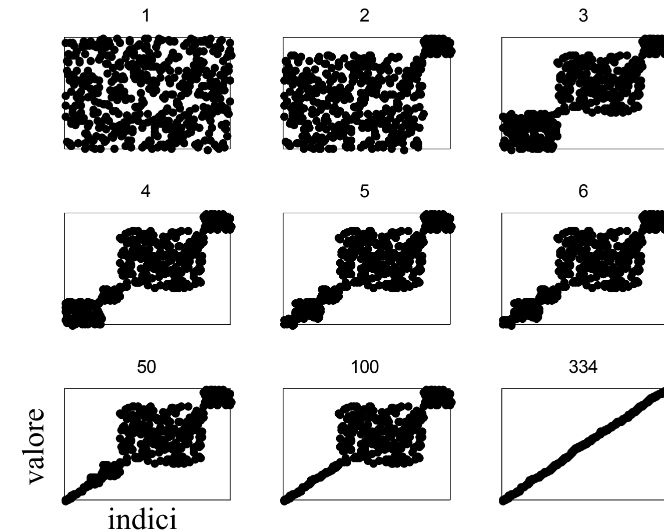


`int[] v={2,5,6,1,3,4};`

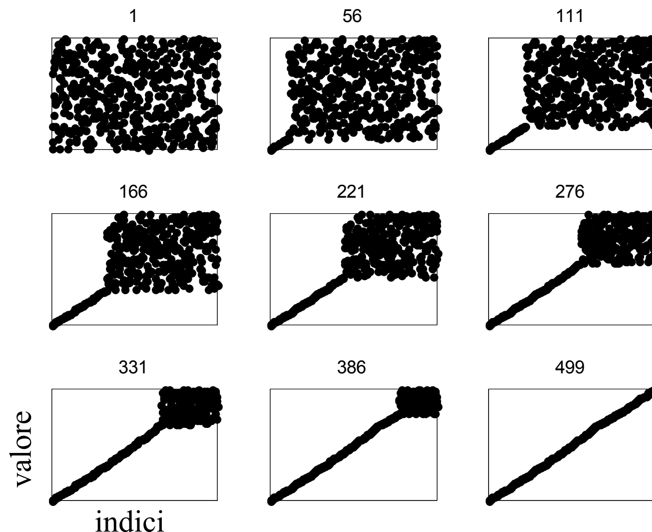
CARATTERISTICHE DINAMICHE

- Vediamo l'ordinamento di un array di 500 elementi, ogni elemento è un valore intero casuale scelto nell'intervallo [0,999].
- I grafici rappresentano la disposizione dei valori dell'array ad uno specifico passo del *processo di ordinamento*, il passo è indicato sopra ogni grafico.

ORDINAMENTO VELOCE



ORDINAMENTO PER SELEZIONE



TIPI DI DATI ASTRATTI

- Il processo di scrittura del codice dovrebbe essere preceduto da uno *schema* del programma con le sue specifiche.
- Fin dall'inizio è importante specificare ciascun compito in termini di *ingresso* e *uscita*.
- Il *comportamento* del programma è più importante dei meccanismi che lo realizzano. Se è necessario un certo dato per realizzare alcuni obiettivi, tale dato è specificato in termini delle operazioni che vengono svolte su esso, piuttosto che della sua struttura interna.

TIPI DI DATI ASTRATTI

- Un tipo di dato specificato mediante le operazioni possibili su di esso è detto *tipo di dato astratto* (*Abstract DataType, ADT*).
- In Java un tipo di dato astratto può far parte di un programma sotto forma di *interfaccia*.
- Le interfacce sono simili alle classi, ma contengono solo le firme dei metodi, non la loro implementazione.
- I metodi vengono definiti dalla classe che realizza (*implementa*) l'interfaccia.

TIPI DI DATI ASTRATTI

- Un ADT è un tipo di dato accessibile *solo* attraverso un'interfaccia. Si definisce *client* un programma (classe) che usa un ADT e si definisce *implementazione* una classe che specifica il tipo di dato.

- Il vantaggio risiede nella possibilità di modificare la rappresentazione dei dati senza modificare i client che la usano.

TIPI DI DATI ASTRATTI : esempio

- Un punto è caratterizzato, per esempio, dalle sue coordinate *cartesiane, polari* e da un'operazione che calcola la *distanza* da un altro punto.
- Lo si può definire come ADT nel modo seguente:

```
public interface Point {
 double x();
 double y();
 double r();
 double theta();
 double distanza(Point p);
}
```


TIPI DI DATI ASTRATTI : esempio

- Rappresentazione interna dei dati in coordinate cartesiane

```
public class Point_im1 implements Point {
  private double x,y;
  public Point_im1(double x, double y){
        this.x=x;
        this.y=y;}
  public double x() {return x;}
  public double y() {return y;}
  public double r() {return Math.sqrt(x*x+y*y);}
  public double theta() {return Math.atan2(y,x);}
  public double distanza(Point p) {
        double dx = this.x() - p.x();
        double dy = this.y() - p.y();
        return Math.sqrt(dx*dx+dy*dy);}
  }
```

TIPI DI DATI ASTRATTI : esempio

- Rappresentazione interna dei dati in coordinate polari

```
public class Point_im2 implements Point {
  private double r ,theta;
  public Point_im2(double x, double y){
        r=Math.sqrt(x*x+y*y);
        theta=Math.atan2(y,x);}
  public double x() {return r*Math.cos(theta);}
  public double y() {return r*Math.sin(theta);}
  public double r() {return r;}
  public double theta() {return theta;}
  public double distanza(Point p) {
        double dx = this.x() - p.x();
        double dy = this.y() - p.y();
        return Math.sqrt(dx*dx+dy*dy);}
}
```


TIPI DI DATI ASTRATTI : esempio

- Sfruttando il concetto di ADT, qualsiasi rappresentazione interna dei dati (cartesiana o polare), non modifica l'uso che ne fanno i client, perchè il comportamento non cambia.

```
public class Test {
  public static void main(String[] args){
      Point_im1 p1 = new Point_im1(1,1);   // cartesiana
      Point_im2 p2 = new Point_im2(1,1);   // polare
      System.out.println("im1: "+ p1.r());  // uso
      System.out.println("im2: "+ p2.r());  // uso
}}
```

im1: 1.4142135623730951
im2: 1.4142135623730951

TIPI DI DATI ASTRATTI : esempio

- La ragione di modificare la rappresentazione dei dati è quella di ottenere *prestazioni migliori*.
- Per esempio, se nei client il metodo `r()` è usato con frequenza, si ottengono prestazioni migliori usando l'implementazione in coordinate polari.
- L'interfaccia di un ADT definisce un "contratto" tra utenti e implementatori che impiega precisi strumenti di comunicazione fra i due contraenti.