Insegnamento di Laboratorio di algoritmi e strutture dati

Interfaccia Comparable e confronto tra QuickSort e MergeSort

Roberto Posenato

ver. 0.7, 29/01/2008

Introduzione

Scopo dell'esercitazione: implementare due algoritmi di ordinamento (QuickSort e MergeSort) per ordinare insiemi di oggetti.
Struttura lezione:

- Introduzione interfaccia `Comparable`;
- Implementazione degli algoritmi di ordinamento QuickSort e MergeSort usando l'interfaccia Comparable;
- Implementazione di una classe di Test per misurare sperimentalmente le prestazioni delle due implementazioni.

Nota!

Si assume che lo studente già conosca i due algoritmi da un punto di vista funzionamento e conosca l'analisi del loro costo computazionale.
Tali argomenti sono già stati svolti nel modulo di teoria.

Interfaccia Comparable

Introduzione

- Si vuole dare un'implementazione degli algoritmi di ordinamento in grado di operare su oggetti (`Object`) anziché su tipi primitivi (`int, char, ecc`).
- Il problema principale quindi è decidere come confrontare gli oggetti.
- Che tipo di confronto si deve utilizzare tra due oggetti x e y?
  - Operatore ==: $x == y$ è vero sse i due oggetti sono il medesimo. È troppo forte. Non utile per lo scopo!
  - sarebbe meglio avere x "uguale" y sse rappresentano lo stesso oggetto, quindi...
  - il confronto deve essere basato sul valore e non sul riferimento!

Interfaccia Comparable

Introduzione

- La classe `Object` implementa metodo `equals()`
  - realizza l'operatore == tra reference;
  - è il candidato ideale per l'overriding (si può ridefinire cosa significa ==);
  - ma non è sufficiente per dire se $x >$ o $< y$.
- è necessario quindi che gli oggetti mettano a disposizione un metodo per confrontarli;
- solo attraverso questo metodo è possibile eseguire un ordinamento.

Interfaccia Comparable

Definizione

- Un modo per richiedere un metodo che esegua un confronto tra oggetti è imporre un'interfaccia.
- `java.lang.Comparable` è una semplice interfaccia che richiede di implementare un solo metodo
  `int compareTo(Object)`
  che deve stabilire se l'oggetto corrente è minore/uguale/maggiore dell'oggetto parametro.

Nota!

- Una classe che implementa in modo corretto l'interfaccia `Comparable` garantisce che le sue istanze (oggetti) sono confrontabili fra loro.
- Un metodo che necessita di confrontare un insieme di oggetti fra loro può facilmente ottenere l'obiettivo richiedendo che gli oggetti siano di tipo `Comparable`.

Interfaccia Comparable

Definizione

Sintesi contratto di Comparable

```
/**
 * Impone un ordine totale sugli oggetti della classe che
 * la implementa. Questo ordinamento è detto anche
 * ordinamento naturale della classe.
 * Un ordinamento è detto consistente con l'uguaglianza
 * sse e1.compareTo(e2) == 0 ha valore identico a
 * e1.equals(e2) per qualsiasi e1 e e2 della classe C.
 *
 * Si noti che null non è un'istanza di alcuna classe, e
 * che e.compareTo(null) dovrebbe lanciare a
 * NullPointerException perfino se e.equals(null)
 * restituisce false.
 * Si raccomanda di implementare il metodo in modo
 * consistente.
 */
public interface Comparable {
```

Interfaccia Comparable

Definizione

continua sintesi contratto di Comparable

```
    /**
     * Restituisce un intero < 0 o 0 o > 0 a seconda che
     * questo oggetto sia <, ==, > dell'argomento.
     * Si deve garantire che sgn(x.compareTo(y)) ==
     * -sgn(y.compareTo(x)).
     * Questo implica che x.compareTo(y) deve lanciare
     * un'eccezione se y.compareTo(x) la lancia.
     * È raccomandato, ma non strettamente richiesto, che
     * (x.compareTo(y)==0) == (x.equals(y)).
     * Se non si soddisfa questa condizione, si deve
     * indicare nel Javadoc che la classe ha un ordine
     * naturale NON consistente con equals().
     */
     int compareTo(Object o);
}
```

Interfaccia Comparable

Esempi

Le seguenti classi implementano `Comparable`

Class	**Natural Ordering**
Byte	signed numerical
Character	unsigned numerical
Long	signed numerical
Integer	signed numerical
Short	signed numerical
Double	signed numerical
Float	signed numerical
BigInteger	signed numerical
BigDecimal	signed numerical
File	system-dep. lexicographic on pathname
String	lexicographic

Interfaccia Comparable

Esempio di costruzione

- Si supponga di avere una classe *A* con due campi:
  1. `int key;`
  2. `String value;`
- Si supponga che l'ordine naturale della classe sia definito come segue. Dati due oggetti, `a` e `b`,
  1. `a` è inferiore a `b` se `a.key < b.key`;
  2. `a` è maggiore a `b` se `a.key > b.key`;
  3. Se `key.a == key.b`, allora si stabilisce l'ordinamento in base all'ordinamento lessicografico di `value`.
- Si vuole completare la classe in modo che implementi Comparable.

Interfaccia Comparable

Esempio di costruzione

Esempio di classe che implementa Comparable

```
public class A implements Comparable {
    ...
    /** ...
     * @see java.lang.Comparable#compareTo()
     */
    public int compareTo(Object o) {
        A o1 = (A) o; //Confronto solo oggetti di A!
        int diff = this.key - o1.key;
        if (diff != 0) return diff;
        return this.value.compareTo(o1.value);
    }
```

Esercizio 1

Individuare l'istruzione che possono sollevare `NullPointerException` e/o `ClassCastException` (documentate nel Javadoc ma non dichiarate). Perché non sono dichiarate?

Interfaccia Comparable

Esempio di costruzione

Continua classe che implementa Comparable

```
/**...
 * @see java.lang.Comparable
 * @see java.lang.Object#equals(java.lang.Object)
 */
public boolean equals(Object o) {
    if (this==o) return true; //È equals di Object
    if (o instanceof A) {//No eccezioni
        A o1 = (A) o;
        return this.key==o1.key &&
            this.value!=null &&
            this.value.equals(o1.value);
    }
    return false;
}
```

Implementazione QuickSort

Richiamo idea algoritmo

1. Dato un array $A[0, \ldots, n]$,
2. si sceglie un elemento (*pivot*),
3. si divide l'array in due parti, una contiene gli elementi $<$ *pivot*, l'altra contiene elementi $\geq$ *pivot*,
4. si iterano i passi 2-3 sulle due parti fino a quando le parti hanno dimensione $= 1$
5. si restituisce l'array A (ordinamento crescente)

Implementazione QuickSort

Osservazioni

- algoritmo di tipo divide et impera (ricorsivo);
- migliori prestazioni quando chiamate ricorsive agiscono su sottoarray di pari lunghezza (circa)
- scelta *pivot* cruciale; diverse strategie:
  - + semplice: scegliere primo elemento array
  - + cauta: scegliere elemento posto al centro array
  - + indipendente: scegliere un elemento casuale array
  - + precisa: scegliere l'elemento mediana array

Implementazione QuickSort

Versione cauta

Quicksort versione cauta

```
/**
 * Ordina il vettore a in modo crescente
 * secondo l'algoritmo QuickSort di Hoare (1961)
 *
 * @param a array da ordinare
 */
public static void QuickSort(Comparable a[]) {
    quickSort(a, 0, a.length - 1);
}
```

Implementazione QuickSort

Versione cauta

continua Quicksort versione cauta

```
/**
 * Implementazione algoritmo ricorsivo QuickSort.
 *
 * @param a array da ordinare
 * @param lower indice inizio sottoarray
 * @param upper indice fine sottoarray
 */
private static void quickSort(Comparable a[],
    int lower, int upper) {
    if (lower < upper) {
        int pivotIndex =
          prudentPartition(a, lower, upper);
        quickSort(a, lower, pivotIndex);
        quickSort(a, pivotIndex + 1, upper);
    }
}
```

Implementazione QuickSort

Versione cauta

continua Quicksort versione cauta

```
/**
 * Permuta i valori di a compresi tra lower e upper
 * rispetto all'elemento pivot: tutti i valori minori di
 * pivot vanno alla sua sx mentre tutti gli altri sua dx.
 * Il pivot è l'elemento che sta in centro all'array.
 * ...
 */
private static int prudentPartition(
  Comparable a[], int lower, int upper) {
    int half = (lower + upper) / 2;
    Comparable pivot = a[half];
    //Si sposti pivot al primo posto
    a[half] = a[lower];
    a[lower] = pivot;
    // inizio procedura di partizionamento
    Comparable temp = null;
    ...
```

Implementazione QuickSort

Versione cauta

continua Quicksort versione cauta

```
    while (true) {
        //Ricerca elemento ≤ pivot da dx.
        while (a[upper].compareTo(pivot) > 0)
            upper--;
        //Ricerca elemento ≥ pivot da sx.
        while (a[lower].compareTo(pivot) < 0)
            lower++;
        if (lower < upper) {
            //Trovati, si scambiano
            temp = a[lower];
            a[lower++] = a[upper];
            a[upper--] = temp;
        } else
            //Finito, si ritorna indice di pivot
            return upper;
    }
}
```

Implementazione QuickSort

Esempio

- $a =$

0	1	2	3	4
D	**M**	**H**	**K**	**B**

- $pivot = H \Rightarrow$

0	1	2	3	4
H	**M**	**D**	**K**	**B**

- I ciclo:
  - Subito prima di `if` ($lower < upper$):

0	1	2	3	4
H	**M**	**D**	**K**	**B**

  - Dopo `if`:

0	1	2	3	4
B	**M**	**D**	**K**	**H**

- II ciclo:
  - Subito prima di `if` ($lower < upper$):

0	1	2	3	4
B	**M**	**D**	**K**	**H**

  - Dopo `if`:

0	1	2	3	4
B	**D**	**M**	**K**	**H**

  - Return 1.

Implementazione QuickSort

Richiamo complessità

Complessità:

- caso migliore: quando pivot è mediana $\Rightarrow O(n \log_2 n)$
- caso peggiore: quando pivot è min o max $\Rightarrow O(n^2)$
- caso medio: più vicino al caso migliore $\Rightarrow O(n \log_2 n)$

Osservazioni:

- può essere reso più efficiente sostituendo ricorsione con iterazione;
- se $n < 30$, QuickSort è più lento di InsertionSort (Cook&Kim 1980);
- in media QuickSort è migliore degli altri algoritmi di ordinamento di un fattore 2.

Implementazione MergeSort

Richiamo idea algoritmo

1. Dato un array $A[0, \ldots, n]$,
2. lo si partiziona ricorsivamente in 2 sottoparti di dimensioni uguali fino a quando la dimensione è 1 (divide),
3. si "fondono" le sottoparti a due a due ordinando gli elementi fino ad riottenere la dimensione originale (impera).
4. si restituisce l'array A (ordinamento crescente).

Implementazione MergeSort

Richiamo idea algoritmo

Fondere due sottoarray in una array ordinato è il task principale.
Semplice procedura di merge!

Procedura `merge`(A, B, C)

// A array destinazione, B, C sottoarray: 3 array!

1: $i_A = i_B = i_C = 0$;
2: **while** *(sia B che C contengono elementi)* **do**
3: $\quad A[i_A{+}{+}] = (B[i_B] < C[i_C])\ ?\ B[i_B{+}{+}]\ :\ C[i_C{+}{+}]$;
4: **endw**
5: inserisci in A tutti elementi rimasti di B o C;
6: **return** A;

Implementazione MergeSort

Richiamo idea algoritmo

Nota!

- B e C sono sottoarray di A: B è la prima metà, C l'altra.
- Per fare la fusione è necessario un array temporaneo!

Procedura di merge di due sottoarray di A

Procedura `merge`$(A, first, last)$

// A array, first: inizio B, last: fine C

1: $mid = (first + last)/2, i_A = 0, i_B = first, i_C = mid + 1;$
2: **while** $(i_B < mid \ \&\& \ i_C < last)$ **do**
3: $\quad temp[i_A{+}{+}] = (A[i_B] < A[i_C]) \ ? \ A[i_B{+}{+}] : A[i_C{+}{+}];$
4: **endw**
5: Inserisci in $temp$ tutti elementi rimasti ($A[i_B, \ldots, mid]$ o $A[i_C, \ldots, last]$);
6: copia $temp$ in $A[first, \ldots, last]$;
7: **return** A;

Implementazione MergeSort

Richiamo idea algoritmo

Nota!

- L'array ausiliario è penalizzante quando $n >> 0$:
  - Occupa spazio pari all'array da ordinare;
  - Si deve poi ricopiare nell'array principale
- Esistono diversi modi per gestire array ausiliario:
  - Più usato: anziché copiare in *temp* e poi ricopiare in A, si può copiare i sottoarray in 2 *temp* di dimensioni mezze e poi fondere in A.
- In ogni caso è opportuno allocare lo spazio per questo array una sola volta!

Implementazione MergeSort

Codice Java

Prima di iniziare, si riporta il Javadoc di un metodo usato nel codice

System.arraycopy()

```
/**
 * Copies an array from the specified source array,
 * beginning at the specified position,
 * to the specified position of the destination array.
 * A subsequence of array components are copied from
 * the source array referenced by src to the destination
 * array referenced by dest.
 * The number of components copied is equal to the length
 * argument.
 * ...
 */
void arraycopy(Object src, int srcPos,
    Object dest, int destPos, int length);
```

Implementazione MergeSort

Codice Java

```
/**
 * Ordina il vettore a in modo crescente
 * secondo l'algoritmo MergeSort.
 *
 * @param a array da ordinare
 */
public static void MergeSort(Comparable a[]) {
    int half = a.length / 2 + 1;
    Object[] leftA = new Object[half],
             rightA = new Object[half];
    mergeSort(a, 0, a.length - 1, leftA,
   rightA);
}
```

Implementazione MergeSort

Codice Java

continua...

```
/**
 * ...
 * @param lo indice inferiore da dove ordinare
 * @param hi indice superiore dove finire ordinare
 * @param leftA, right A sottoarray di supporto
 */
private static void mergeSort(Comparable a[],
        int lo, int hi,
        Object[] leftA, Object[] rightA) {

    if (lo >= hi) return; //Recursion base

    //Partition a into two lists and sort them recursively
    int middle = (lo + hi) / 2;
    mergeSort(a, lo, middle, leftA, rightA);
    mergeSort(a, middle + 1, hi, leftA, rightA);
    ...
```

Implementazione MergeSort

Codice Java

continua...

```
    //Merge the two sorted lists
    int left = 0,              right = 0,
        sizeL = middle-lo+1, sizeR = hi-middle;

    System.arraycopy(a, lo, leftA, 0, sizeL);
    System.arraycopy(a, middle+1, rightA, 0,
    sizeR);
    while ((left < sizeL) && (right < sizeR))
      a[lo++]=((Comparable) leftA[left])
                .compareTo(rightA[right]) <= 0
              ? (Comparable) leftA[left++]
              : (Comparable) rightA[right++];
    while (left < sizeL)
      a[lo++] = (Comparable) leftA[left++];
    while (right < sizeR)
      a[lo++] = (Comparable) rightA[right++];
}
```

Implementazione MergeSort

Esempio

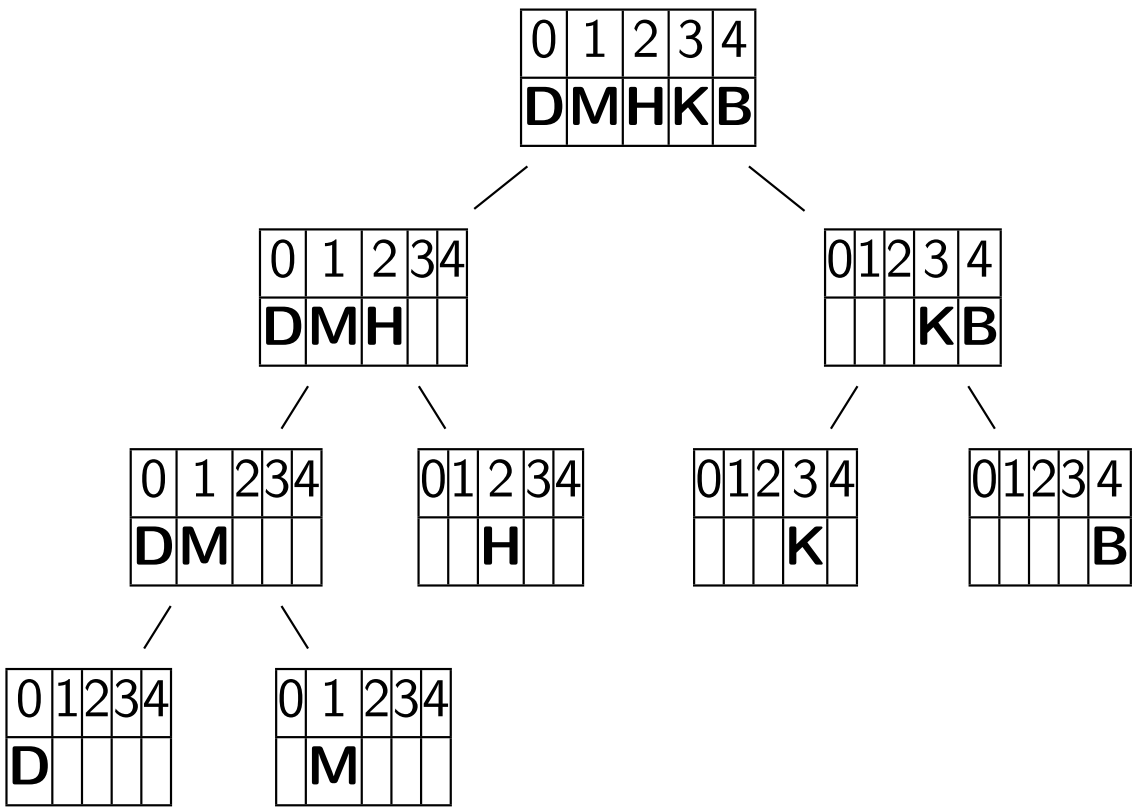
- $a =$

0	1	2	3	4
D	**M**	**H**	**K**	**B**

- $leftA =$ | 0 | 1 | 2 |
|---|---|---|
| | | |

$rightA =$ | 0 | 1 | 2 |
|---|---|---|
| | | |

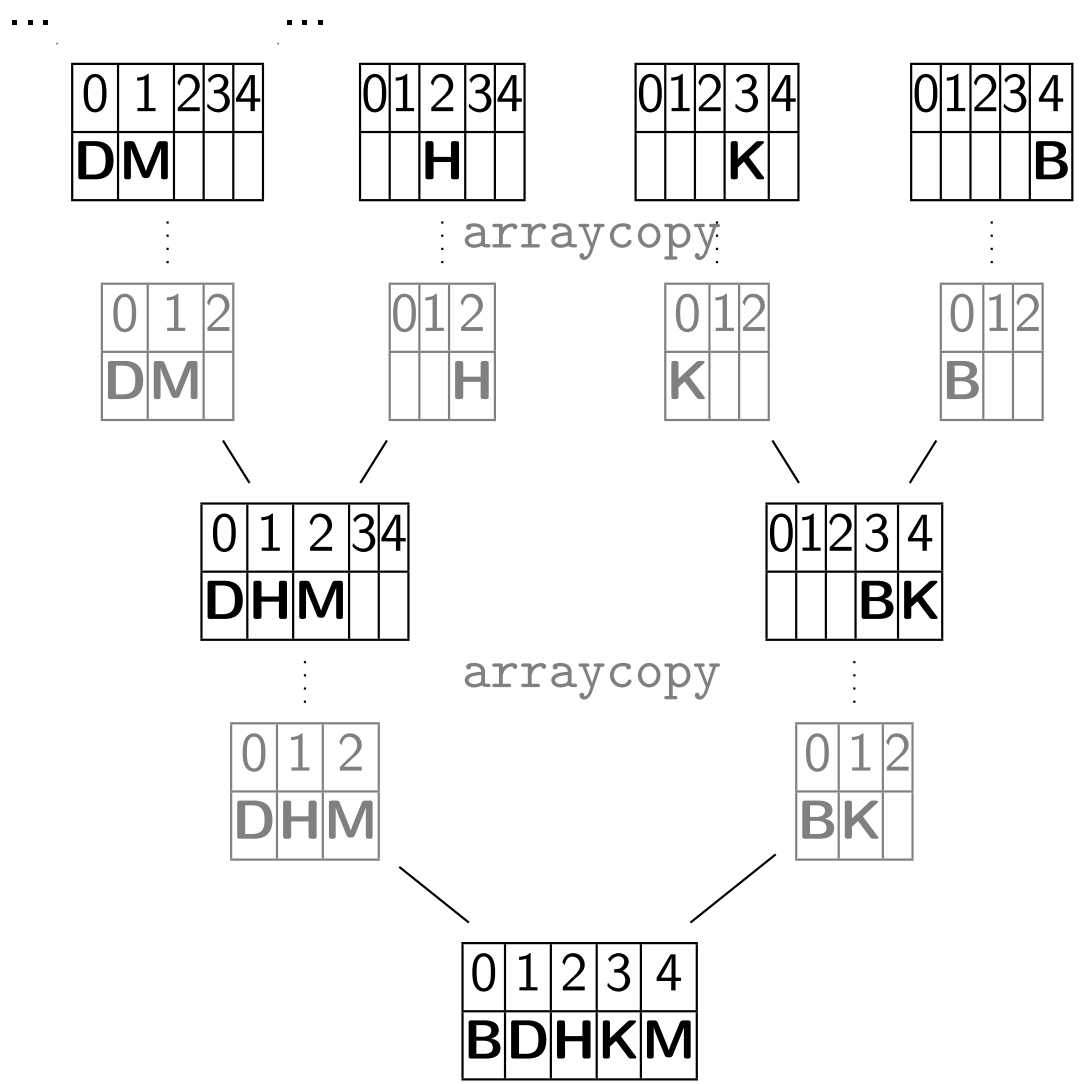
- Fase divide:



Implementazione MergeSort

Esempio

- Fase merge:



Implementazione MergeSort

Richiamo complessità

Complessità:

- caso migliore: quando a è in ordine o elementi *leftA* precedono elementi *rightA* $\Rightarrow O(n \log_2 n)$
- caso peggiore: quando ultimo elemento di *leftA* precede solo ultimo elemento di *rightA* $\Rightarrow O(n \log_2 n)$

Osservazioni:

- può essere reso più efficiente sostituendo ricorsione con iterazione o applicando InsertionSort quando array ha dimensioni piccole;
- la richiesta di spazio aggiuntivo penalizza questo algoritmo;

Confronto sperimentale tra QuickSort e MergeSort

Obiettivo

Si vuole confrontare i tempi di esecuzione delle implementazioni costruite:

- si genera un array casuale;
- si applicano i due algoritmi sul medesimo array rilevando i tempi di calcolo;

Nota!

I tempi di calcolo possono essere falsati dall'iterazione del programma con sistema operativo.
Una possibilità è calcolare il tempo per eseguire la medesima operazione (ordinamento) più volte e considerare come stima del tempo il rapporto fra il tempo totale e il numero di ripetizoni eseguite.

Confronto sperimentale tra QuickSort e MergeSort

Classe Random per TestSort

Prima di iniziare, si riporta il Javadoc di una classe usata nel codice.

java.util.Random

```
/**
 * An instance of this class is used to generate a stream
 * of pseudorandom numbers.
 * ...
 */
public class Random {
    ...
    /**
     * Returns a pseudorandom, uniformly distributed int
     * value between 0 (inclusive) and the specified value
     * (exclusive)....
     */
    public int nextInt(int n);
```

Confronto sperimentale tra QuickSort e MergeSort

Classe TestSort

```
public class TestSort {
  public static void main(String[] args) {
    ...
    //Definizione variabili
    Random  rnd = new Random();
    Integer[] a = new Integer[n]
              , b = new Integer[n];

    //Si riempiono i due vettori con valori casuali
    for (i = 0; i < n; i++) {
      a[i]=b[i]=new Integer(rnd.nextInt(n*2));
    }
    ...
```

Confronto sperimentale tra QuickSort e MergeSort

Classe TestSort

```
        //Tempo per ordinare gli elementi con MergeSort
        for (i = 0, tempo = 0; i < nCicli; i++) {
          inizio = System.currentTimeMillis();
          Sort.MergeSort(a);
          fine = System.currentTimeMillis();
          tempo += (fine - inizio);
          //Si ripristina l'array
          System.arraycopy(b, 0, a, 0, n);
        }
        System.out.println(
          "Tempo richiesto (ms) da MergeSort: "
          + (tempo / nCicli)
        );
        ...
    }
```

Algoritmi di ordinamento

Librerie Java

Nel linguaggio Java esistono già delle implementazioni di alcuni algoritmi di ordinamento.

Per array

La classe `java.util.Arrays` contiene diversi metodi basati sul QuickSort o MergeSort per i diversi tipi di array:

- `static void sort(int[] a); //char,double,...`
- `static void sort(Object[] a);//Object <-> Comparable`

Algoritmi di ordinamento

Librerie Java

continua 'Per array'

- `static void sort(Object[] a, Comparator c);`
  - Permette di ordinare array in base a criteri diversi (c) e di avere elementi nulli nell'array;
  - utilizza l'interfaccia `java.util.Comparator` che ha un solo metodo:

```
/**
 * Compares its two arguments for order.
 * Returns a negative integer, zero, or a positive integer
 * as the first argument is less than, equal to,
 * or greater than the second.
 * ...
 */
int compare(Object o1, Object o2);
```

Algoritmi di ordinamento

Librerie Java

Per `liste`

La classe `java.util.Collections` contiene due metodi basati sul MergeSort:

- `static void sort(List l);`
- `static void sort(List l, Comparator c);`

Algoritmi di ordinamento

Esercizio

Esercizio 2

- Confrontare i tempi di calcolo di QuickSort, MergeSort e Array.sort per $n = 10000, 20000, 40000, 80000$.
- Stampare la tabella con i coefficienti di crescita (**CC**) del tempo per ogni algoritmo.
  Esempio: tempi in ms su Intel(R) Xeon(TM) CPU 3.20GHz con Linux 2.6

N.	CC	Merge	CC	Quick	CC	Arrays.sort	CC
10 000	-	38	-	4	-	28	-
20 000	2	45	1,18	7	1,75	33	1,17
40 000	2	53	1,18	15	2,14	39	1,18
80 000	2	76	2,43	32	2,13	47	1,21