Laboratorio di Algoritmi e Strutture dati

10 aprile 2015

⋆ Esercizio `sort`: algoritmi di ordinamento *(10 pt)*

Si crei un package *sort*, all'interno del quale si definiscano una classe *Sorting* insieme alla sua classe di test, e inoltre due classi *SortTiming* e *FastSortTiming*, come specificato nel seguito.

Sorting (raccolta di algoritmi di ordinamento)

In un file *Sorting.java* si definisca una omonima classe non istanziabile, e priva di main, contenente metodi statici pubblici che realizzano i diversi algoritmi di ordinamento studiati nel corso, per array di `int` oppure generici, eventualmente in più versioni, oltre ovviamente a tutti i metodi ausiliari necessari. Si definisca inoltre un metodo `isSorted` che controlla se un array è ordinato.

Esempio:

```
// restituisce true se l'array è ordinato, false se non lo è
public static bool isSorted(int[] a)

public static <T extends Comparable<? super T>>
  bool isSorted(T[] a)

public static void ssort(int[] a)

public static <T extends Comparable<? super T>>
  void ssort(T[] a)

public static void isort(int[] a)

public static <T extends Comparable<? super T>>
  void isort(T[] a)

// insertion sort con uso della ricerca binaria del punto di
    inserimento
public static void isortBin(int[] a)

void msortBasic(int[] a)
```

```
// con unico array ausiliario e merge ottimizzato, slides
    19.53 e 19.56
public static void msortNoGarbage(int[] a)

public static <T extends Comparable<? super T>>
  void msortNoGarbage(T[] a)

// versione "a passo alternato", (slide 19.60)
public static void msortAlt(int[] a)
public static <T extends Comparable<? super T>>
  void msortAlt(T[] a)

// msort ottimizzato con isort sotto una data soglia
public static void msortIsort(int[] a)

public static <T extends Comparable<? super T>>
  void msortIsort(T[] a)

// versione parallela di una delle versioni del msort che
    avete scelto di realizzare prima
public static void parallelMergesort(int[] a)

...

// quicksort in una versione base, cioé "estraendo il pivot"
void qsortBasic(int[] a)

public static <T extends Comparable<? super T>>
  void qsortBasic(T[] a)

// quicksort in una versione alle Hoare, cioè "senza
    estrarre esplicitamente il pivot"
void qsortHoare(int[] a)

public static <T extends Comparable<? super T>>
  void qsortHoare(T[] a)

// quicksort in una versione alle Hoare, ottimizzato con
    isort, eventualmente con eliminazione della ricorsione
    di coda
void qsortHoareIsort(int[] a)


...

// versione parallela di una delle versioni del quicksort
    che avete scelto di realizzare prima
public static void parallelQuicksort(int[] a)
```

```
...

// heapsort, studiato nella seconda parte del corso
public static <T extends Comparable<? super T>>
  void hsort(T[] a)

// introspective sort}, studiato nella seconda parte del
    corso
public static <T extends Comparable<? super T>>
  void introsort(T[] a)

...
```

In particolare, si realizzino almeno:

- una versione del ssort;
- una versione dell'isort;
- una versione sequenziale del mergesort;
- una versione sequenziale del quicksort;
- una versione parallela del mergesort;
- una versione parallela del quicksort;
- una dello heapsort (algoritmo studiato in una fase del corso successiva a quella iniziale).

Nel commento ad ogni metodo si indichi chiaramente quale versione dell'algoritmo viene realizzata, e quali sono le sue caratteristiche. La presenza, accanto ad una versione base di ogni algoritmo, di una versione più raffinata ed efficiente sarà valutata positivamente.

SortTiming

In un file *SortTiming.java* si definisca una omonima classe non istanziabile, contenente un main e tutti gli opportuni metodi ausiliari. Il programma esegue gli algoritmi di ordinamento realizzati in `Sorting`, ne misura i tempi di esecuzione e produce un file `.csv` che riporta i tempi rilevati. Il programma deve effettuare misurazioni su array di dimensioni crescenti distinte per gli algoritmi quadratici e per quelli ottimali. Gli algoritmi quadratici vanno musurati insieme ad almeno un algoritmo ottimale (al fine di apprezzarne la differenza di prestazioni). In entrambi i casi gli ultimi esperimenti (quelli che misurano i tempi di esecuzione su array di dimensioni maggiori) devono forzare gli algoritmi a lavorare per almeno qualche decina di secondi.
Si testino infine i metodi sort e parallelSort della classe Arrays della libreria di Java, che hanno un'ottimizzazione molto spinta e sono quindi presumibilmente i più veloci di tutti.

Nota: Al fine di semplificare la scrittura del programma di misurazione dei tempi verrà fornito un'implementazione di esempio da completare a cura dello studente.