

Laboratorio di ASD

Primo progetto

a.a. 2006-2007

Gabriele Savio - Michele Di Cosmo

Indice

1. Introduzione	3	3.3 MergeSort	9
2. Implementazione del programma di raccolta e analisi dei dati	3	3.4 QuickSort	9
2.1 Generazione di numeri casuali	3	3.5 HeapSort	10
2.2 Input (classe)	3	4. Implementazione di SuperSort	10
2.2.1 getTaraRip – Calcolo delle ripetizioni della tara	4	5. Tabelle	11
2.2.2 getTaraTime – Calcolo del tempo medio della tara	4	5.1 BubbleSort	11
2.3 Algoritmi (classe astratta)	4	5.2 InsertionSort	11
2.3.1 getRipLordo – Calcolo delle ripetizioni lorde	4	5.3 MergeSort	12
2.3.2 executeTest – Calcolo del tempo lordo	5	5.4 QuickSort	12
2.4 Funzionamento generale (classe Tester)	5	5.5 HeapSort	13
2.4.1 getGranularita – Calcolo della granularità δ	5	5.6 SuperSort	13
2.4.2 Ciclo di test	5	6. Grafici e discussione sui confronti della complessità	14
2.5 Gerarchia	7	6.1 BubbleSort vs. InsertionSort	14
2.6 Rilevazione dei tempi	8	6.2 MergeSort vs. QuickSort	14
2.7 Costanti	8	6.3 InsertionSort vs. QuickSort vs. SuperSort	15
3. Descrizione ed analisi degli algoritmi	9	6.3.1 QuickSort vs. SuperSort	15
3.1 BubbleSort	9	6.3.2 Zoom sull'intersezione	16
3.2 InsertionSort	9	6.4 MergeSort vs. QuickSort vs. HeapSort	16
		7. Discussione dei risultati ottenuti	17
		8. Strumenti utilizzati	17
		9. Conclusioni	18
		Bibliografia	19

1. Introduzione

In questa analisi ci poniamo l'obiettivo di confrontare il comportamento di alcuni algoritmi di ordinamento nel loro caso medio (con input casuale) in relazione alla teoria discussa nel corso.

È stato realizzato un programma in linguaggio Java atto a raccogliere i dati sperimentali ottenuti eseguendo test per stimare il tempo di esecuzione degli algoritmi *BubbleSort*, *InsertionSort*, *MergeSort*, *QuickSort* e *HeapSort* su un vettore di numeri interi di lunghezza 100, 200, 500, 1000, 2000, 5000, 10000.

Per ogni algoritmo abbiamo calcolato il tempo medio di esecuzione su diverse dimensioni di input. Basandoci poi sui risultati così ottenuti abbiamo elaborato un algoritmo denominato *SuperSort* che ottimizza *InsertionSort* e *QuickSort* sfruttando il minor tempo di esecuzione del primo per input di dimensione ridotta e quello del secondo per input di dimensione consistente.

I risultati sono stati elaborati e presentati qui in forma tabellare e grafica dando particolare risalto ai loro andamenti asintotici.

Di seguito introduciamo l'implementazione del programma per la raccolta e l'analisi dei dati sperimentali.

2. Implementazione del programma di raccolta e analisi dei dati

2.1 Generazione di numeri casuali

Per la generazione dei numeri casuali abbiamo sfruttato la classe **RandomGenerator** fornita durante il corso.

2.2 Input (classe)

L'input casuale è stato realizzato tramite la classe **Input** istanziabile con i parametri *dimension*, che specifica la lunghezza dell'array, e *rndGen*, che rappresenta una istanza di **RandomGenerator** (adibita alla generazione di numeri casuali).

Si noti che in questo modo la classe **RandomGenerator** viene istanziata un'unica volta all'avvio del programma fornendo input diversi senza la necessità di modificare il seme su istanze diverse.

La classe **Input** fornisce i metodi `copy`, che restituisce un array `int[]` contenente una copia dell'input rappresentato dalla classe, `getTaraRip`, che restituisce il numero di ripetizioni necessarie per calcolare il tempo di tara e `getTaraTime`, che restituisce il tempo effettivo di tara.

2.2.1 getTaraRip – Calcolo delle ripetizioni della tara

Il metodo `getTaraRip` calcola, attraverso il codice analizzato durante il corso, il numero di ripetizioni necessarie ad effettuare l'operazione `copy` sull'input, ovvero a copiare i dati su un vettore `int[]`, in modo che siano dell'ordine di grandezza che ci permetta di avere la precisione desiderata.

2.2.2 getTaraTime – Calcolo del tempo medio della tara

Il metodo `getTaraTime` calcola il tempo necessario a eseguire `rip` volte l'operazione `copy` sull'input, quindi divide il tempo totale per il numero di ripetizioni (`rip`) restituendo il tempo medio dell'operazione `copy`.

2.3 Algoritmi (classe astratta)

L'implementazione degli algoritmi si è basata sullo pseudo-codice elaborato durante il corso di Laboratorio. Comunque, è stato necessario adattare lo pseudo-codice al linguaggio Java. In particolare gli indici di riferimento agli array sono stati modificati da 1-based¹ a 0-based.

E' stata realizzata una classe astratta **Algoritmo** dalla quale tutti gli algoritmi ereditano.

Algoritmo mette a disposizione i metodi `getRipLordo` per il calcolo delle ripetizioni lorde di un particolare algoritmo, `execute` per l'esecuzione unitaria dell'algoritmo ed `executeTest` per l'esecuzione di `rip` volte dell'algoritmo. In quest'ultimo caso viene restituito il tempo lordo per l'esecuzione unitaria dell'algoritmo tramite `execute` (*tempo totale* – `rip`) con `rip` calcolato da `getRipLordo`.

Algoritmo mette a disposizione alle classi che lo ereditano le operazioni comuni degli algoritmi: `swap` e `partition`.

Gli algoritmi veri e propri (i.e. **BubbleSort**, **MergeSort**, ...) specificano obbligatoriamente i metodi `execute` (`int[] input`) per l'esecuzione dell'algoritmo.

2.3.1 getRipLordo – Calcolo delle ripetizioni lorde

Il metodo `getRipLordo` calcola il numero di ripetizioni necessarie ad eseguire l'operazione `copy` su **Input** e `execute` su **Algoritmo** affinché siano dell'ordine di grandezza che ci permetta di avere la precisione desiderata.

¹ Per 1-based si intende un vettore il cui indice iniziale è 1 (i.e. i vettori nella convenzione del pseudo-codice). Per 0-based un vettore con indice iniziale 0 (i.e. i vettori in Java).

2.3.2 executeTest – Calcolo del tempo lordo

Il metodo `executeTest` esegue rip volte l'operazione `copy` su **Input** e `execute` su **Algoritmo** restituendo il tempo medio per l'esecuzione delle due.

2.4 Funzionamento generale (classe Tester)

La classe **Tester** è adibita a eseguire i test e presentare i risultati all'utente.

Vengono eseguite le seguenti operazioni di inizializzazione:

- a) Impostazione del vettore degli algoritmi e delle dimensioni su cui il test verrà eseguito e istanziazione degli algoritmi
- b) Calcolo della granularità δ tramite il metodo `getGranularita`
- c) Calcolo di $tMin$ (vedi ²)
- d) Creazione di una istanza di **RandomGenerator**
- e) Calcolo della tara per ogni dimensione di input tramite i metodi `getTaraRip` e `getTaraTime` di **Input**

Terminate tutte le operazioni di inizializzazione viene eseguito il ciclo di test (vedi par. 2.4.2), adibito al calcolo vero e proprio dei dati.

2.4.1 getGranularita – Calcolo della granularità δ

Per il calcolo della granularità δ utilizziamo lo pseudo-codice visto durante il corso.

La granularità del sistema è la risoluzione dell'orologio di sistema e dipende dalle caratteristiche della macchina e dal sistema operativo. Rappresenta la risoluzione del nostro sistema di misura.

2.4.2 Ciclo di test

Il ciclo di test è il cuore della raccolta e analisi dei dati sperimentali.

Inizialmente viene eseguito un ciclo sugli algoritmi da testare.

Successivamente viene eseguito un secondo ciclo annidato sulle dimensioni dei vettori di input.

Quindi si entra in un terzo ciclo annidato per eseguire $totInput$ volte le ripetizioni con dimensione di input fissa. All'interno di questo ciclo vengono eseguite le seguenti operazioni:

² Dato che l'errore di misurazione scelto è 5% ($k = 0,05$), bisogna misurare un tempo $t \geq \delta/k$ che chiameremo $tMin = r \cdot \delta$ dove $r = 1/k$. Nel nostro caso avremo $tMin = 20\delta$.

- a) Viene istanziata una classe **Input** contenente numeri casuali
- b) Vengono calcolate le ripetizioni lorde tramite il metodo `getRipLordo` di **Algoritmo**
- c) Viene eseguito l'algoritmo tramite il metodo `executeTest` di **Algoritmo** e calcolato $tLordo$

Per ogni ciclo sulla dimensione viene calcolato $tMedio = \left(\frac{tLordo}{totInput} \right) - \left(\frac{TaraTime}{TaraRip} \right)$ con $TaraTime$ e $TaraRip$ calcolati in precedenza e $\Delta = \frac{1}{\sqrt{totInput}} z \left(1 - \frac{\alpha}{2} \right) s(n)$ (vedi ³) usato per determinare l'intervallo di confidenza.

³ $\alpha = 0,05$ quindi la funzione di distribuzione normale $z \left(1 - \frac{\alpha}{2} \right) = 1,96$.

La varianza campionaria è invece definita come $s(n)^2 = \frac{1}{totInput} \sum_{i=1}^{totInput} X(n, i)^2 - E'[X(n)]^2$.

2.5 Gerarchia

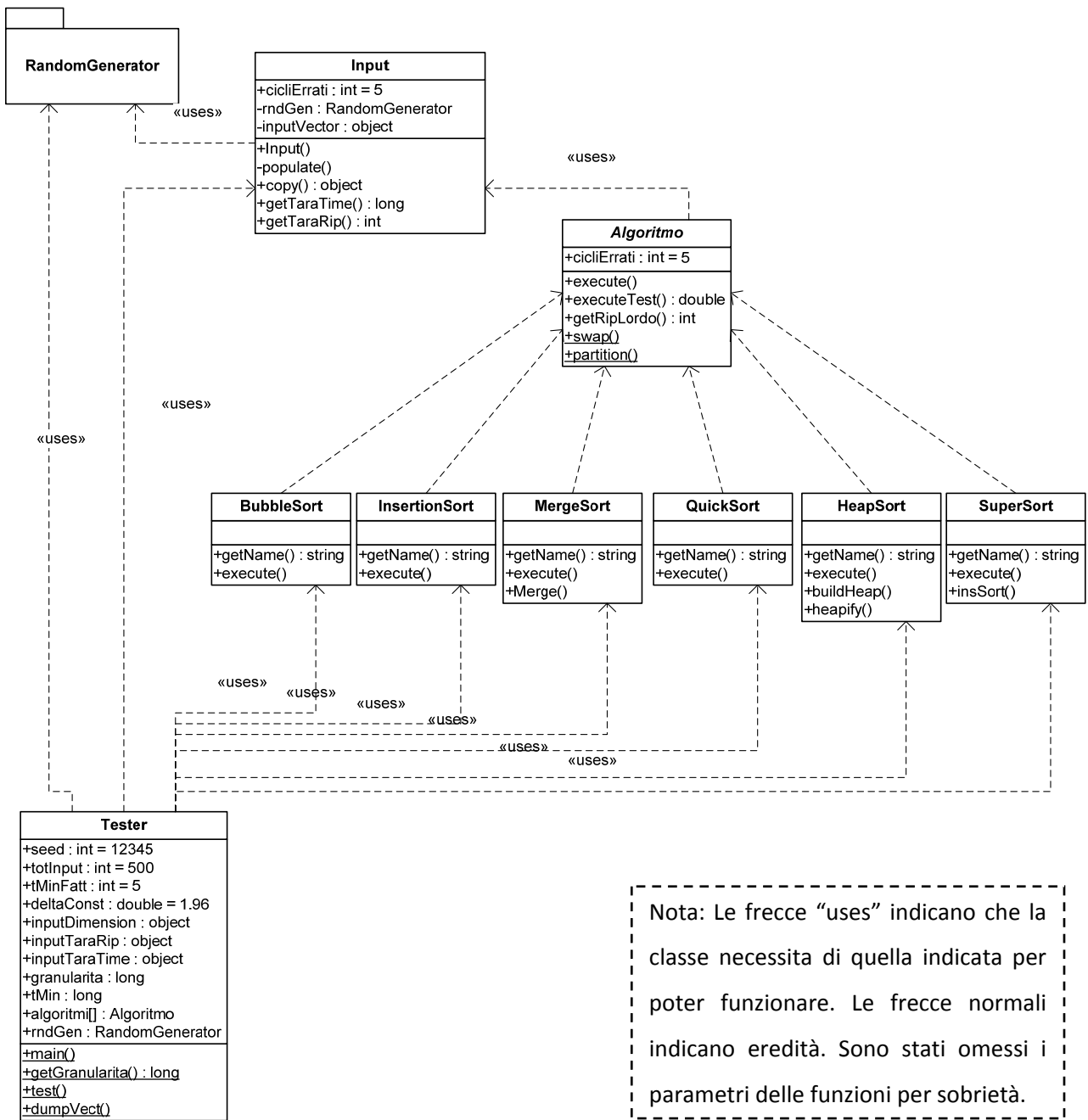


Fig. 1 - Schema UML

Tutti gli algoritmi ereditano dalla classe astratta **Algoritmo**.

Gli algoritmi, gli input e la classe **RandomGenerator** vengono istanziati direttamente dalla procedura main di **Tester**.

Algoritmo necessita di **Input**, **Input** necessita di **RandomGenerator** e **Tester** necessita di tutte le classi.

2.6 Rilevazione dei tempi

La rilevazione dei tempi è stata effettuata tramite la funzione `System.currentTimeMillis()` che restituisce la differenza, misurata in millisecondi, tra il tempo attuale e la mezzanotte del primo gennaio 1970 UTC.

Il tempo viene calcolato dalla funzione `executeTest` memorizzando l'istante in cui viene avviata l'esecuzione di *ripLordo* istanze di input di dimensione uguale. Al termine dell'esecuzione viene sottratto il tempo attuale a quello precedentemente memorizzato, quindi diviso per *ripLordo*. Il procedimento viene ripetuto per *totInput* volte pari alla cardinalità del campione dei dati su input diversi, ma di stessa dimensione. Il tempo totale viene sommato in *tLordo*, quindi diviso per *totInput* in modo da ottenere il tempo lordo per l'esecuzione unitaria dell'algoritmo. Sottraendo il tempo medio della tara al tempo lordo si ottiene *tMedio*.

La scelta di *totInput* è stata determinata dall'osservazione del variare dell'intervallo di confidenza con diversi esperimenti. In conclusione un valore pari a 500 ci ha fornito una stima sufficientemente accurata.

È stato scelto di calcolare il numero di ripetizioni lorde necessarie per ottenere una stima accurata ogni qualvolta venisse istanziato un nuovo input. Questo perché un input generato casualmente potrebbe venir ordinato in un tempo inferiore a quello medio e pertanto rendere inaccurate tutte le rilevazioni effettuate con quel numero di ripetizioni.

2.7 Costanti

Sono state usate le seguenti costanti:

Parametro	Valore	Descrizione
Seed	12345	Seme usato da RandomGenerator
totInput	500	Istanze del problema (campione dei dati)
tMinFatt	5 %	Errore di misurazione minimo ammesso
α	5 %	Esperimenti "cattivi" consentiti

3. Descrizione ed analisi degli algoritmi

3.1 BubbleSort

BubbleSort opera scambiando ripetutamente gli elementi adiacenti che non sono ordinati. (Vedi appendice)

E' un algoritmo in-place e ha complessità nel caso medio $\Theta(n^2)$.

3.2 InsertionSort

InsertionSort opera scandendo gli elementi del vettore e inserendoli uno ad uno creando una porzione ordinata del vettore. (Vedi appendice)

E' un algoritmo in-place efficiente per ordinare un piccolo numero di elementi. Ha complessità nel caso medio $O(n^2)$.

3.3 MergeSort

MergeSort è un algoritmo divide et impera che si divide in tre passi: 1) divide l'array a metà; 2) ordina ricorsivamente i due sotto-array; 3) combina i due sotto-array tramite un'operazione di Merge. (Vedi appendice)

E' un algoritmo non in-place perché la procedura Merge fa uso di un vettore ausiliario. Ha complessità nel caso medio $\Theta(n \log n)$.

3.4 QuickSort

QuickSort è anch'esso un algoritmo divide et impera che si divide in due passi: 1) partiziona l'array tramite la procedura Partition in due sotto-array tali che ogni elemento del primo sia minore o uguale ad ogni elemento del secondo; 2) ordina ricorsivamente i due sotto-array. (Vedi appendice)

E' un algoritmo in-place con complessità nel caso medio $\Theta(n \log n)$.

3.5 HeapSort

HeapSort fa uso della struttura dati heap binario⁴. Opera trasformando l'array di input in una max-heap tramite la procedura Build-Max-Heap quindi isola la radice (elemento massimo) diminuendo la dimensione della heap e ricostruisce la Max-Heap fino a quando essa contiene un solo elemento. (Vedi appendice)

E' un algoritmo in-place perché la heap viene creata sul vettore stesso di input e i numeri ordinati vengono scritti nella parte dell'array non utilizzata dalla heap. Ha complessità nel caso medio $\Theta(n \log n)$.

4. Implementazione di SuperSort

Per definire **SuperSort** abbiamo creato una variante ottimizzata di **QuickSort** che sfrutta la logica dell'algoritmo InsertionSort quando la dimensione della porzione del vettore da ordinare è minore di una costante $k = 23$ ottenuta analizzando sperimentalmente l'andamento di QuickSort e InsertionSort per un intorno di quelle dimensioni (Grafico 5). (Vedi appendice)

Sperimentalmente l'analisi dell'andamento di SuperSort mostra che questa ottimizzazione porta a risultati migliori di QuickSort e InsertionSort (Grafico 5).

⁴ Un heap binario è un albero quasi-completo (dove al più l'ultimo livello non è completo) in cui per ogni nodo i , suo figlio sinistro e suo figlio destro hanno $key(son) \leq key(i)$ se max-heap, $key(son) \geq key(i)$ se min-heap. Gli heap sono facilmente astruibili con un array.

5. Tabelle

5.1 BubbleSort

Dimensione	<i>tMedio</i> (ms)	Δ
100	0,04651132	0,00065345
200	0,19358528	0,00186868
500	1,22663323	0,00751779
1.000	4,88183009	0,02096474
2.000	19,33478251	0,06594697
5.000	120,68626697	0,24930691
10.000	488,23728590	0,80984848

Tabella 1 BubbleSort

5.2 InsertionSort

Dimensione	<i>tMedio</i> (ms)	Δ
20	0,00090130	0,00005229
22	0,00117964	0,00006711
25	0,00211966	0,00009983
27	0,00283034	0,00010082
30	0,00338565	0,00011112
100	0,01692947	0,00067249
200	0,05232697	0,00133433
500	0,31523047	0,00361702
1.000	1,25857539	0,01010853
2.000	5,05570130	0,03253404
5.000	31,53726697	0,12220216
10.000	128,63578590	0,36066356

Tabella 2 InsertionSort

5.3 MergeSort

Dimensione	<i>tMedio</i> (ms)	Δ
100	0,04783971	0,00064583
200	0,10019670	0,00128948
500	0,25924933	0,00319480
1.000	0,54040438	0,00632333
2.000	1,12835039	0,01340325
5.000	2,98157387	0,03350689
10.000	6,44205221	0,07194485

Tabella 3 MergeSort

5.4 QuickSort

Dimensione	<i>tMedio</i> (ms)	Δ
20	0,00112556	0,00006559
22	0,00164684	0,00008800
25	0,00175486	0,00009401
27	0,00154499	0,00007430
30	0,00195073	0,00009638
100	0,00863412	0,00027761
200	0,02042939	0,00058420
500	0,09500950	0,00339026
1.000	0,23962539	0,00700631
2.000	0,51707368	0,01538605
5.000	1,19446648	0,04321511
10.000	2,36016754	0,08689184

Tabella 4 QuickSort

5.5 HeapSort

Dimensione	<i>tMedio</i> (ms)	Δ
100	0,02144417	0,00043223
200	0,04872353	0,00089396
500	0,14086080	0,00233500
1.000	0,30641316	0,00475792
2.000	0,67663084	0,01033735
5.000	1,86855921	0,02666333
10.000	4,05616864	0,05613868

Tabella 5 HeapSort

5.6 SuperSort

Dimensione	<i>tMedio</i> (ms)	Δ
20	0,00089380	0,00004988
22	0,00103655	0,00005460
25	0,00119389	0,00005965
27	0,00130817	0,00006661
30	0,00151031	0,00007182
100	0,00736295	0,00026318
200	0,01761005	0,00053509
500	0,05261323	0,00141684
1.000	0,11814652	0,00301975
2.000	0,26202100	0,00643948
5.000	0,73965388	0,01674467
10.000	1,66480147	0,04378098

Tabella 6 SuperSort

6. Grafici e discussione sui confronti della complessità

6.1 BubbleSort vs. InsertionSort

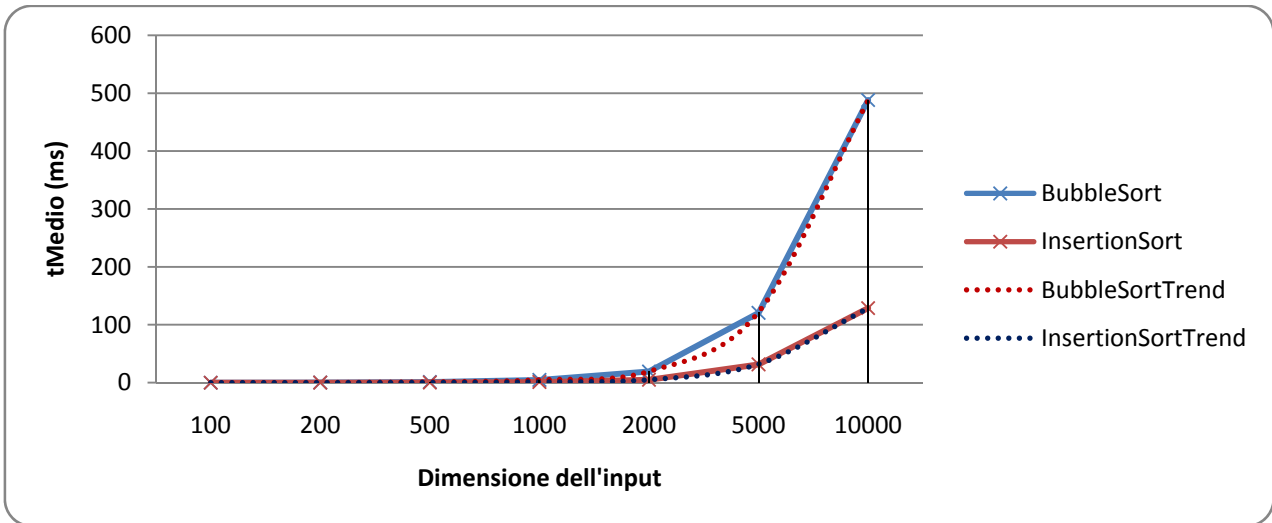


Grafico 1 BubbleSort vs. InsertionSort

	$f(x)$	Errore asintotico standard	% err.
BubbleSortTrend	$f(x) = 4,87907 \cdot 10^{-6} x^2$	$\pm 5,320 \cdot 10^{-9}$	0,10900 %
InsertionSortTrend	$f(x) = 1,28486 \cdot 10^{-6} x^2$	$\pm 2,412 \cdot 10^{-9}$	0,18770 %

6.2 MergeSort vs. QuickSort

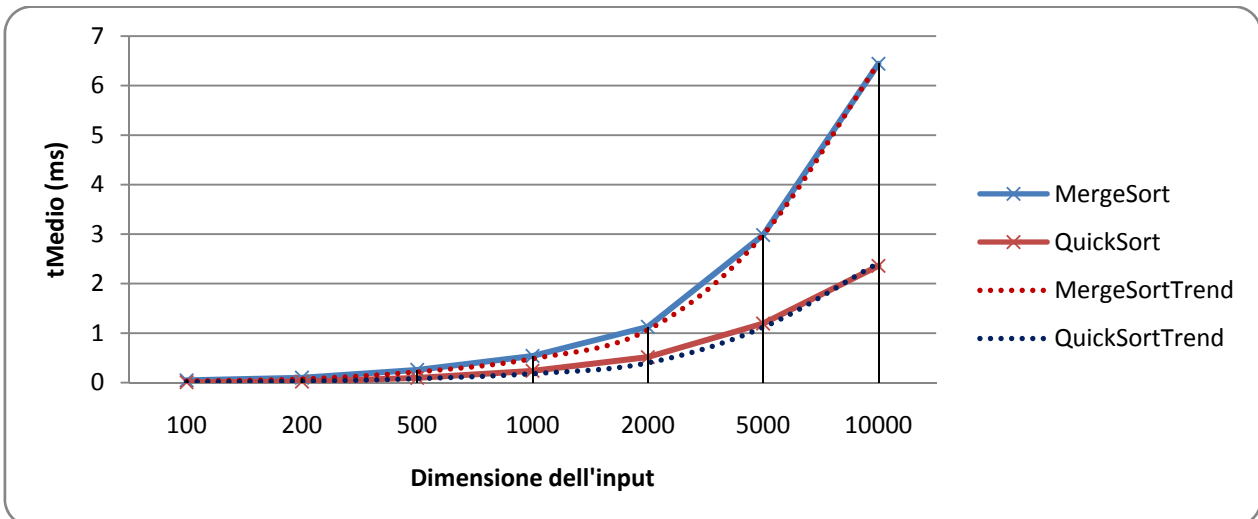


Grafico 2 MergeSort vs. QuickSort

	$f(x)$	Errore asintotico standard	% err.
MergeSortTrend	$f(x) = 4,85910 \cdot 10^{-5} x \cdot \log_2 x$	$\pm 2,742 \cdot 10^{-7}$	0,56420 %
QuickSortTrend	$f(x) = 1,82076 \cdot 10^{-5} x \cdot \log_2 x$	$\pm 4,498 \cdot 10^{-7}$	2,47000 %

6.3 InsertionSort vs. QuickSort vs. SuperSort

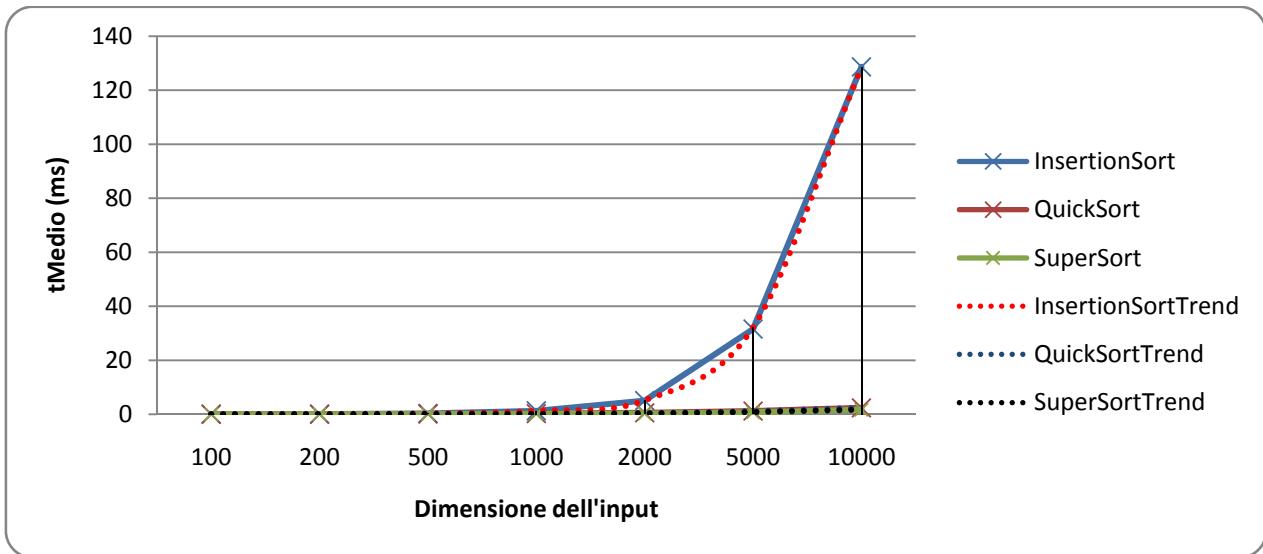


Grafico 3 InsertionSort vs. QuickSort vs. SuperSort

	$f(x)$	Errore asintotico standard	% err.
InsertionSortTrend	$f(x) = 1,28486 \cdot 10^{-6}x^2$	$\pm 2,412 \cdot 10^{-9}$	0,18770 %
QuickSortTrend	$f(x) = 1,82076 \cdot 10^{-5}x \cdot \log_2 x$	$\pm 4,498 \cdot 10^{-7}$	2,47000 %
SuperSortTrend	$f(x) = 1,24282 \cdot 10^{-5}x \cdot \log_2 x$	$\pm 8,284 \cdot 10^{-8}$	0,66650 %

6.3.1 QuickSort vs. SuperSort

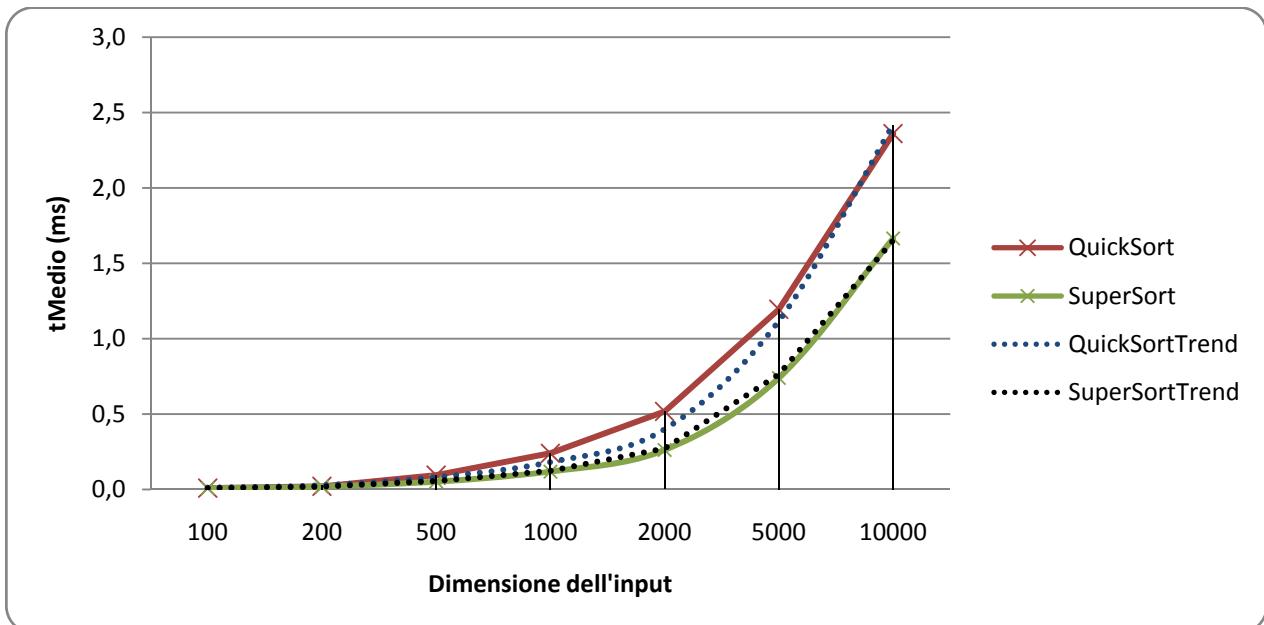


Grafico 4 QuickSort vs. SuperSort

6.3.2 Zoom sull'intersezione

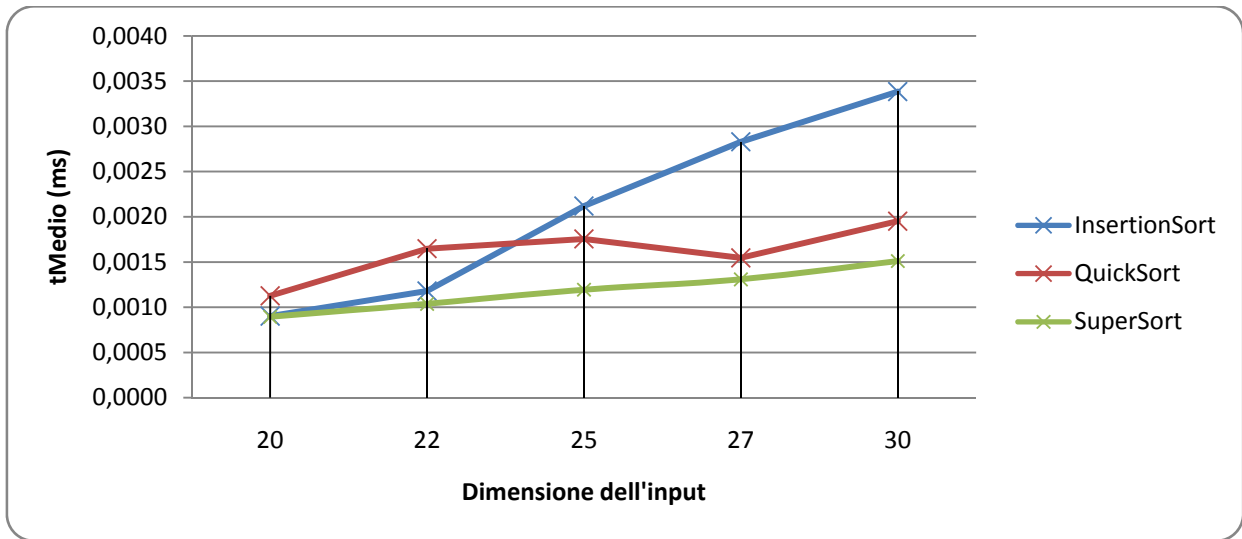


Grafico 5 Zoom di InsertionSort vs. QuickSort vs. SuperSort

6.4 MergeSort vs. QuickSort vs. HeapSort

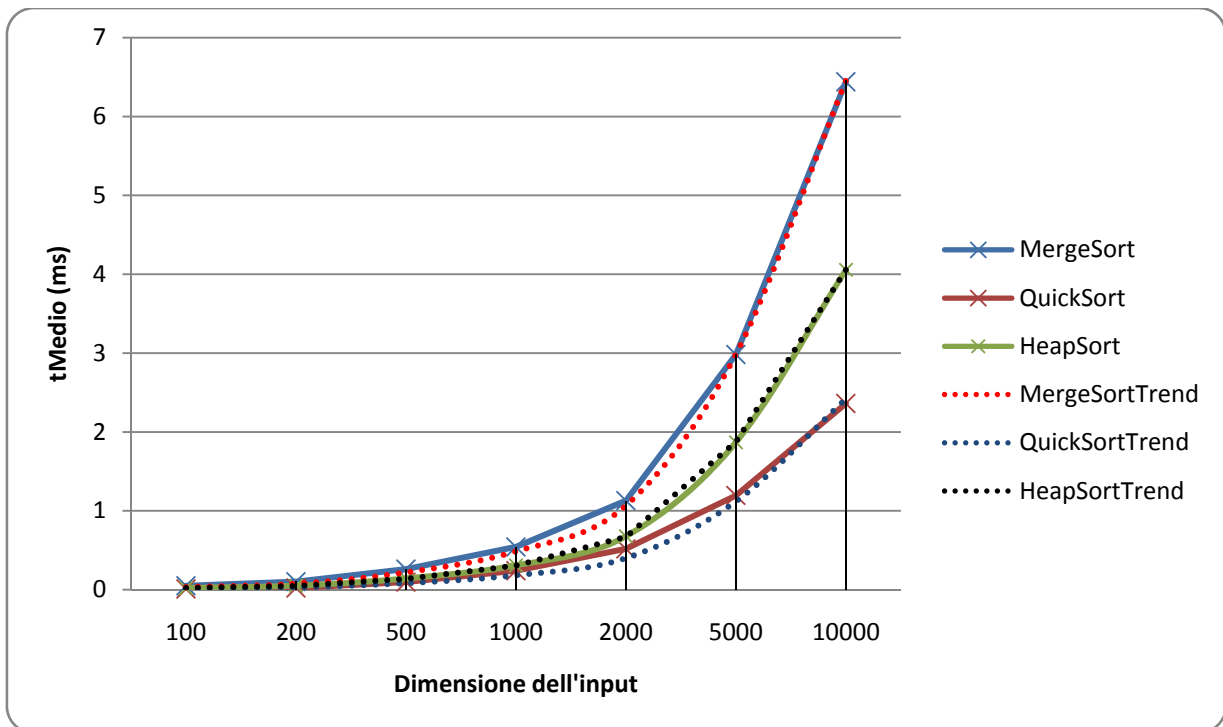


Grafico 6 MergeSort vs. QuickSort vs. HeapSort

	$f(x)$	Errore asintotico standard	% err.
MergeSortTrend	$f(x) = 4,85910 \cdot 10^{-5} x \cdot \log_2 x$	$\pm 2,742 \cdot 10^{-7}$	0,56420 %
QuickSortTrend	$f(x) = 1,82076 \cdot 10^{-5} x \cdot \log_2 x$	$\pm 4,498 \cdot 10^{-7}$	2,47000 %
HeapSortTrend	$f(x) = 3,05156 \cdot 10^{-5} x \cdot \log_2 x$	$\pm 3,051 \cdot 10^{-8}$	0,09998 %

7. Discussione dei risultati ottenuti

Di seguito esponiamo le nostre analisi.

Dal Grafico 1 possiamo confrontare l'andamento dei due algoritmi di complessità quadratica: BubbleSort e InsertionSort. Notiamo che BubbleSort impiega notevolmente più tempo a ordinare input di grandi dimensioni rispetto a InsertionSort; peraltro questo comportamento era ampiamente previsto dagli studi teorici dato che l'algoritmo BubbleSort esegue sempre e comunque n^2 confronti. Al contrario, InsertionSort esegue solo n confronti in caso di elementi già ordinati.

Dal Grafico 2 e dal Grafico 6 possiamo osservare con attenzione l'andamento dei tre algoritmi di complessità $\Theta(n \log n)$. Gli studi teorici non avevano messo in risalto le costanti moltiplicative di tali algoritmi, precludendoci la possibilità di analizzare quale fosse effettivamente il più efficiente.

Dai dati osserviamo che MergeSort risulta il "più lento" fra i tre analizzati, nel senso che richiede più elaborazione per ordinare vettori di grandi dimensioni.

Al contrario QuickSort si pone come la migliore scelta tra i tre ordinando in un tempo minore sia di MergeSort che di HeapSort.

Dopo esserci posti il problema di dover creare un algoritmo che avesse una costante moltiplicativa migliore rispetto agli altri algoritmi studiati, abbiamo deciso di usare i due algoritmi che su dimensioni differenti si comportavano meglio: InsertionSort su piccole dimensioni e QuickSort su grandi dimensioni (Grafico 5). Abbiamo quindi creato SuperSort: una variante di QuickSort che richiama al suo interno InsertionSort quando QuickSort si trova a trattare input di dimensione minore di $k = 23$ (con k ottenuto empiricamente osservando il Grafico 5) (vedasi paragrafo 4, pag. 10). Quindi abbiamo analizzato il comportamento di quest'ultimo che, in relazione a QuickSort e agli altri algoritmi, risulta esser migliore (vedasi paragrafo 6.3.1).

8. Strumenti utilizzati

Per l'esecuzione del programma di raccolta e analisi dei dati sperimentali è stato utilizzato un calcolatore con processore **Intel Pentium 4** con una **frequenza di 3,00 GHz** e **1,00 GB di RAM**. La macchina è stata

eseguita su una clean-install⁵ del sistema operativo è **Windows XP Home Edition con Service Pack 2** su cui sono stati installati **J2SDK 1.4.2.13**, **JavaSE-RE 6** e **BlueJ 2.1.3**.

Per la post-elaborazione dei dati sono stati usati **Microsoft Excel 2007**.

Le funzioni interpolanti sono state ottenute con il metodo $f_i t$ di **gnuPlot 4.2.0** compilato su piattaforma win32. Tale metodo utilizza un'implementazione dell'algoritmo NLLS ("nonlinear least-squares Marquardt-Levenberg algorithm").

Per la presentazione dei dati sono stati usati **Microsoft Word 2007** e **Microsoft Visio 2007**.

9. Conclusioni

In conclusione possiamo dire che lo studio sperimentale degli algoritmi basati su confronto studiati durante il corso, ci ha permesso di confrontarci con molte problematiche sull'implementazione degli algoritmi, sulla rilevazione dei tempi infinitesimali del calcolatore e su una realizzazione gerarchica flessibile del codice.

Inoltre, queste problematiche ci hanno aiutato a comprendere il valore di una buona analisi statistica, come ad esempio la scelta di un buon campione di esperimenti.

Abbiamo compreso l'importanza di scegliere un buon algoritmo adeguato alle proprie esigenze e di migliorarlo in base alla situazione specifica.

Concludendo possiamo affermare che le analisi sperimentali si sono rivelate complementari e necessarie agli studi teorici.

⁵ Per clean-install si intende un sistema operativo appena installato senza applicativi aggiuntivi.

Bibliografia

- Introduzione agli algoritmi e strutture dati (T. H. Cormin, C. E. Leiserson, R. L. Rivest, C. Stein)
McGraw-Hill – 2° edizione
- Java – fondamenti di progettazione software (J. Lewis, W. Loftus) Addison-Wesley – 1° edizione
- Dispense del corso di Laboratorio di algoritmi e strutture dati (S. Scalabrin) 2007
- Appunti del corso di Algoritmi e strutture dati

```

1  /** Questa classe contiene il codice per eseguire i test sugli algoritmi
2  * e presentare all'utente i risultati
3  *
4  *
5  * @author (G. Savio, M. Di Cosmo)
6  * @version (23/04/2007 - 23.48)
7  */
8
9  public class Tester
10 {
11     public static final int seed = 12345; // Seme per la generazione di vettori con
numeri casuali
12     public static final int totInput = 500; // Numero di volte che l'algoritmo viene
eseguito su un dato input
13     public static final int tMinFatt = 5; // % Errore ammesso
14     public static final double deltaConst = 1.96; // Costante di delta, vedi tabella.
15     public static int[] inputDimension; // Dimensione degli input
16     public static int[] inputTaraRip; // N° di ripetizioni della tara per una
determinata dimensione di input
17     public static long[] inputTaraTime; // Tempo della tara per una determinata
dimensione di input
18     public static long granularita; // Granularità
19     public static long tMin; // tMin
20     public static Algoritmo algoritmi[]; // Array di algoritmi su cui verrà eseguito
il test
21     public static RandomGenerator rndGen;
22
23     public static void main(String[] args){
24         main();
25     }
26
27     public static void main(){
28
29         long startTime = System.currentTimeMillis();
30
31         System.out.println ("");
32         System.out.println (">");
33
34         /** Modificare le seguenti impostazioni per ottenere i risultati voluti **/
35
36         // Imposto le dimensioni degli input
37         inputDimension = new int[7];
38         inputDimension[0] = 100;
39         inputDimension[1] = 200;
40         inputDimension[2] = 500;
41         inputDimension[3] = 1000;
42         inputDimension[4] = 2000;
43         inputDimension[5] = 5000;
44         inputDimension[6] = 10000;
45
46         // Imposto gli algoritmi
47         algoritmi = new Algoritmo[6];
48         algoritmi[0] = new SuperSort();
49         algoritmi[1] = new QuickSort();
50         algoritmi[2] = new HeapSort();
51         algoritmi[3] = new MergeSort();
52         algoritmi[4] = new InsertionSort();
53         algoritmi[5] = new BubbleSort();
54
55         /** Fine sezione impostazioni utente **/
56
57         /** Init **/
58         // Calcolo la granularità
59         granularita = getGranularita();
60         System.out.println ("% Granularità = " + granularita);
61
62         // Calcolo tMin
63         long tMinGranCoeff = (long) ((float)1 / ((float)tMinFatt / (float)100));
64         tMin = granularita * tMinGranCoeff;
65         System.out.println ("% Errore ammesso = " + tMinFatt + "%");
66         System.out.println ("% tMin = " + tMin + " (granularità * " + tMinGranCoeff + ")
");
67         System.out.println ("% Costante di delta = " + deltaConst);
68
69         rndGen = new RandomGenerator(seed);

```

```

70     System.out.println ("% Seme = " + seed);
71
72     System.out.println ("% Ripetizione degli algoritmi su input diversi = " +
totInput);
73
74     // Scrivo le dimensioni
75     System.out.print ("% Dimensioni =");
76     for (int i = 0; i < inputDimension.length; i++){
77         System.out.print (" " + inputDimension[i]);
78     }
79     System.out.println ("");
80
81     // Scrivo gli algoritmi
82     System.out.print ("% Algoritmi =");
83     for (int i = 0; i < algoritmi.length; i++){
84         System.out.print (" " + algoritmi[i].getName());
85     }
86     System.out.println ("");
87
88     /** Tara **/
89     // Calcolo la tara degli input
90     inputTaraRip = new int[inputDimension.length];
91     inputTaraTime = new long[inputDimension.length];
92     for (int i = 0; i < inputDimension.length; i++){
93         System.out.print ("Calcolo della tara per la dimensione " + inputDimension
[i] + "...");
94
95         Input input = new Input(rndGen, inputDimension[i]);
96
97         inputTaraRip[i] = input.getTaraRip(tMin);
98         System.out.print (" rip=" + inputTaraRip[i]);
99
100        inputTaraTime[i] = input.getTaraTime(inputTaraRip[i]);
101        System.out.println (" time=" + inputTaraTime[i]);
102
103    }
104
105    System.out.println ("");
106    System.out.println ("* Tabella tara");
107    System.out.println ("idx\tdim\ttaraRip\ttaraTime");
108    for (int i = 0; i < inputDimension.length; i++){
109        System.out.println (i + "\t" + inputDimension[i] + "\t" + inputTaraRip[i] +
"\t" + inputTaraTime[i]);
110    }
111
112    /** Algoritmi **/
113    System.out.println ("");
114    System.out.println ("* Tabella esecuzione algoritmi");
115    System.out.println ("idxAlg\tidxDim\tAlg.\tDim.\t1°ripLordo\ttLordo\ttLordo^2\
ttMedio\tDelta\tTime");
116    for (int alg = 0; alg < algoritmi.length; alg++){
117        /** Dimensioni **/
118        for (int dim = 0; dim < inputDimension.length; dim++){
119
120            System.out.print (alg);
121            System.out.print ("\t" + dim);
122            System.out.print ("\t" + algoritmi[alg].getName());
123            System.out.print ("\t" + inputDimension[dim]);
124
125            // Istanzio e azzero tLordo
126            double tLordo = 0;
127            double tLordoQuadrato = 0;
128
129            long totInputTime = System.currentTimeMillis();
130
131            /** Input diversi eseguiti ripLordo volte **/
132            for (int nInput = 0; nInput < totInput; nInput++){
133
134                // Genero l'input
135                Input input = new Input(rndGen, inputDimension[dim]);
136
137                // Calcolo ripLordo
138                int ripLordo = algoritmi[alg].getRipLordo(tMin, input);
139
140                if (nInput == 0){

```

```

141         System.out.print ("\t" + ripLordo);
142     }
143
144     // Aggiungo il tempo lordo impiegato a tLordo
145     double tAlg = algoritmi[alg].executeTest(input, ripLordo);
146     tLordo = tLordo + tAlg;
147     tLordoQuadrato = tLordoQuadrato + (tAlg * tAlg);
148
149     }
150     tLordo = tLordo / totInput;
151     tLordoQuadrato = tLordoQuadrato / totInput;
152     System.out.print ("\t" + tLordo);
153
154     System.out.print ("\t" + tLordoQuadrato);
155
156     // tMedio
157     double tMedio = (tLordo - ((double)inputTaraTime[dim] / (double)
inputTaraRip[dim]));
158     System.out.print ("\t" + tMedio);
159
160     // Delta
161     double Delta = (1 / Math.sqrt(totInput) * deltaConst * (Math.sqrt
(tLordoQuadrato - (tMedio * tMedio))));
162     System.out.print ("\t" + Delta);
163
164     System.currentTimeMillis();
165     System.out.print ("\t" + (System.currentTimeMillis() - totInputTime));
166
167     System.out.println ("");
168
169     }
170 }
171
172 // Fine
173 System.out.println ("");
174 System.out.println ("< Terminato (" + (System.currentTimeMillis() - startTime)
+ "ms");
175
176 }
177
178 public static void Test(int seed){
179
180     System.out.println ("");
181     System.out.println ("Test started with seed " + seed);
182
183     rndGen = new RandomGenerator(seed);
184
185     Input input = new Input(rndGen, 1000);
186     int[] vect = input.copy();
187
188     System.out.print ("before: ");
189     dumpVect(vect);
190
191     Algoritmo alg = new QuickSort();
192     System.out.print ("executing " + alg.getName() + "...");
193
194     long t0 = System.currentTimeMillis();
195     alg.execute(vect);
196     long t1 = System.currentTimeMillis();
197     System.out.println (" done in " + (t1-t0) + "ms");
198
199     System.out.print ("after: ");
200     dumpVect(vect);
201
202     System.out.println ("Checking vector...");
203     int value = vect[0];
204     for (int i = 0; i < vect.length; i++){
205         if (value > vect[i]){
206             System.out.println ("! ERROR ! @" + i);
207             return;
208         }else{
209             value = vect[i];
210         }
211     }
212

```

```

213     System.out.println ("Test ended");
214     System.out.println ("");
215 }
216
217 public static long getGranularita(){
218     // Calcola la granularità
219
220     long t0 = System.currentTimeMillis();
221     long t1 = System.currentTimeMillis();
222
223     while (t0 == t1){
224         t1=System.currentTimeMillis();
225     }
226
227     return (t1 - t0);
228
229 }
230
231 private static void dumpVect(int[] vect){
232     System.out.print ("VECT: [" + vect.length + "]);
233     for (int i = 0; i < vect.length; i++){
234         System.out.print (" " + vect[i]);
235     }
236     System.out.println ("");
237 }
238
239 }
240

```

```

1 /**
2  * Questa classe astratta implementa le funzioni di base di un algoritmo
3  * per il test
4  *
5  * @author (G. Savio, M. Di Cosmo)
6  * @version (23/04/2007 - 23.48)
7  */
8
9 public abstract class Algoritmo
10 {
11
12     public static final int cicliErrati = 5;
13
14     public abstract String getName();
15
16     // Implementa l'esecuzione vera e propria dell'algoritmo
17     public abstract void execute(int[] input);
18
19     public double executeTest(Input input, int rip){
20         // Esegue l'algoritmo sull'input passato rip volte
21         // Viene restituito tLordo
22
23         double t0 = System.currentTimeMillis();
24
25         for (int i = 0; i < rip; i++){
26             //System.out.println ("esecuzione " + i + " / " + rip);
27             execute(input.copy());
28         }
29
30         double t1 = System.currentTimeMillis();
31         //System.out.println (" Tempo su un ciclo rip da " + rip + ": " + (t1 - t0) + "ms");
32
33         return (t1 - t0) / rip;
34     }
35 }
36
37 public int getRipLordo(long tMin, Input input){
38     // Calcola ripLordo per questo algoritmo
39
40     long t0 = 0;
41     long t1 = 0;
42     int rip = 1;
43     int[] vect = input.copy();
44
45     while (t1 - t0 <= tMin){
46
47         rip = rip * 2;
48         t0 = System.currentTimeMillis();
49
50         for (int i = 1; i <= rip; i++){
51             execute(input.copy());
52         }
53
54         t1 = System.currentTimeMillis();
55     }
56
57     int max = rip;
58     int min = rip / 2;
59
60     while (max - min >= cicliErrati){
61
62         rip = (max + min) / 2;
63         t0 = System.currentTimeMillis();
64
65         for (int i = 1; i <= rip; i++){
66             execute(input.copy());
67         }
68
69         t1 = System.currentTimeMillis();
70
71         if (t1 - t0 <= tMin){
72             min = rip;
73         }else{
74             max = rip;
75         }

```

```

76     }
77
78     return max;
79
80 }
81
82 protected static void swap(int[] vett, int i, int j){
83     // Scambia l'elemento i con j nel vettore vett
84     int temp=vett[i];
85     vett[i]=vett[j];
86     vett[j]=temp;
87 }
88
89 protected static int partition(int[] vett, int p, int r){
90     // Esegue la procedura partition sul vettore vett dall'indice p all'indice r
91     int x = vett[r];
92     int i = p - 1;
93     for (int j = p; j < r; j++){
94         if (vett[j] <= x){
95             i++;
96             swap(vett, i, j);
97         }
98     }
99     swap(vett, i + 1, r);
100    return i + 1;
101 }
102
103 }
104

```

```

1 /**
2  * Implementa l'algoritmo BubbleSort
3  *
4  * @author (G. Savio, M. Di Cosmo)
5  * @version (23/04/2007 - 23.48)
6  */
7
8 public class BubbleSort extends Algoritmo
9 {
10
11     public String getName(){
12         return "BubbleSort";
13     }
14
15     public void execute (int[] input){
16
17         int n = input.length - 1;
18
19         for (int j = n; j > 0; j--){
20             for (int i = 0; i < j; i++){
21                 if (input[i] > input[i+1]){
22                     swap(input, i, i+1);
23                 }
24             }
25         }
26
27     }
28
29 }
30

```

```

1 /**
2  * Implementa l'algorithm MergeSort
3  *
4  * @author (G. Savio, M. Di Cosmo)
5  * @version (23/04/2007 - 23.48)
6  */
7
8 public class MergeSort extends Algoritmo
9 {
10
11     public String getName(){
12         return "MergeSort";
13     }
14
15     public void execute(int[] input){
16         execute(input, 0, input.length - 1);
17     }
18
19     public void execute(int[] input, int p, int r){
20         if (p < r){
21             int q = (int) Math.floor ((p + r) / 2);
22             execute(input,p,q);
23             execute(input,q+1,r);
24             Merge(input,p,q,r);
25         }
26     }
27
28     private void Merge(int[] vett, int p, int q, int r){
29         int i = p;
30         int j = q + 1;
31         int[] tempvett = new int[r - p + 1];
32
33         for (int k = 0; k < r - p + 1; k++){
34             if ((i <= q) && (j <= r)){
35                 if (vett[i] < vett[j]){
36                     tempvett[k] = vett[i];
37                     i++;
38                 }else{
39                     tempvett[k] = vett[j];
40                     j++;
41                 }
42             }else{
43                 if(i>q){
44                     tempvett[k]=vett[j];
45                     j++;
46                 }else{
47                     tempvett[k]=vett[i];
48                     i++;
49                 }
50             }
51         }
52         for(int k=0;k<r-p+1;k++){
53             vett[p+k]=tempvett[k];
54         }
55     }
56 }
57 }

```

```

1 /**
2  * Implementa l'algoritmo InsertionSort
3  *
4  * @author (G. Savio, M. Di Cosmo)
5  * @version (23/04/2007 - 23.48)
6  */
7
8 public class InsertionSort extends Algoritmo
9 {
10
11     public String getName(){
12         return "InsertionSort";
13     }
14
15     public void execute (int[] input){
16
17         int n = input.length;
18
19         for (int j = 1; j < n; j++){
20             int temp = input[j];
21             int i = j - 1;
22             while ((i >= 0) && (input[i] > temp)){
23                 input [i + 1] = input [i];
24                 i--;
25             }
26             input [i + 1] = temp;
27         }
28
29     }
30
31 }

```

```

1 /**
2  * Implementa l'algoritmo HeapSort
3  *
4  * @author (G. Savio, M. Di Cosmo)
5  * @version (23/04/2007 - 23.48)
6  */
7
8 public class HeapSort extends Algoritmo
9 {
10
11     private int heapsize = 0;
12
13     public String getName(){
14         return "HeapSort";
15     }
16
17     public void execute(int[] input){
18         buildHeap(input);
19         for (int i = input.length - 1; i > 0; i--){
20             swap(input, 0, i);
21             heapsize--;
22             heapify(input, 0);
23         }
24     }
25
26     private int left(int i){return 2*i;}
27     private int right(int i){return 2*i+1;}
28     private int parent(int i){return (int)Math.floor(i/2);}
29
30     private void buildHeap(int[] vett){
31         heapsize = vett.length - 1;
32         for (int i = (int)Math.floor((vett.length - 1) / 2); i >= 0; i--){
33             heapify(vett, i);
34         }
35     }
36
37     private void heapify(int[] vett, int i){
38         int max;
39         int l = left(i);
40         int r = right(i);
41         if ((l <= heapsize) && (vett[l] > vett[i])){
42             max = l;
43         }else{
44             max = i;
45         }
46         if ((r <= heapsize) && (vett[r] >= vett[max])){
47             max = r;
48         }
49         if (max != i){
50             swap(vett, i, max);
51             heapify(vett, max);
52         }
53     }
54 }
55 }

```

```

1 /**
2  * Implementa l'algoritmo MergeSort
3  *
4  * @author (G. Savio, M. Di Cosmo)
5  * @version (23/04/2007 - 23.48)
6  */
7
8 public class QuickSort extends Algoritmo
9 {
10
11     public String getName(){
12         return "QuickSort";
13     }
14
15     public void execute(int[] input){
16         execute(input, 0, input.length - 1);
17     }
18
19     public void execute(int[] input, int p, int r){
20         if (p < r){
21             int q = partition (input, p, r);
22             execute(input, p, q-1);
23             execute(input, q + 1, r);
24         }
25     }
26 }
27 }

```

```

1 /**
2  * Implementa l'algorithm SuperSort
3  *
4  * @author (G. Savio, M. Di Cosmo)
5  * @version (23/04/2007 - 23.48)
6  */
7
8 public class SuperSort extends Algoritmo
9 {
10
11     public static final int k = 23;
12
13     public String getName(){
14         return "SuperSort";
15     }
16
17     public void execute(int[] input){
18         execute(input, 0, input.length - 1);
19     }
20
21     public void execute(int[] input, int p, int r){
22         if (r - p <= k){
23             insSort(input,p,r);
24         }else{
25             if (p < r){
26                 int q = partition(input, p, r);
27                 execute(input, p, q - 1);
28                 execute(input, q + 1, r);
29             }
30         }
31     }
32
33     private void insSort(int[] input,int p,int r){
34         for (int j = p; j <= r; j++){
35             int temp = input[j];
36             int i = j - 1;
37             while ((i >= 0) && (input[i] > temp)){
38                 input[i + 1] = input[i];
39                 i--;
40             }
41             input[i + 1] = temp;
42         }
43     }
44 }
45 }

```

```

1 /**
2  * Rappresentazione di un input vettoriale
3  *
4  * @author (G. Savio, M. Di Cosmo)
5  * @version (23/04/2007 - 23.48)
6  */
7
8 public class Input
9 {
10
11     public static final int cicliErrati = 5;
12     private RandomGenerator rndGen;
13
14     private int[] inputVector;
15
16     public Input(RandomGenerator rndGen, int dimension){
17         // Crea un array di dimensione dimension
18         this.rndGen = rndGen;
19         inputVector = new int[dimension];
20         populate();
21     }
22
23     private void populate(){
24         // Popola l'array con valori casuali
25         for (int i = 0; i < inputVector.length; i++){
26             inputVector[i] = (int) Math.round (rndGen.get() * inputVector.length);
27         }
28     }
29
30     public int[] copy(){
31         // Restituisce un input in formato int[] duplicato da quello rappresentato
32         dalla classe
33         int[] ret = new int[inputVector.length];
34         for (int i = 0; i < inputVector.length; i++){
35             ret[i] = inputVector[i];
36         }
37         return ret;
38     }
39
40     public long getTaraTime(int tareRip){
41         // Restituisce il tempo di tara per questo input
42
43         long t0 = System.currentTimeMillis();
44
45         for (int i = 0; i < tareRip; i++){
46             copy();
47         }
48
49         long t1 = System.currentTimeMillis();
50
51         return (t1 - t0);
52     }
53
54     public int getTaraRip(long tMin){
55         // Restituisce le ripetizioni minime per la tara
56
57         long t0 = 0;
58         long t1 = 0;
59
60         int rip = 1;
61
62         while (t1 - t0 <= tMin){
63
64             rip = 2 * rip;
65             t0 = System.currentTimeMillis();
66
67             for (int i = 1; i <= rip; i++){
68                 copy();
69             }
70
71             t1 = System.currentTimeMillis();
72         }
73
74         int max = rip;

```

```
75     int min = rip / 2;
76
77     while (max - min >= cicliErrati){
78
79         rip = (max + min) / 2;
80         t0 = System.currentTimeMillis();
81
82         for (int i = 1; i <= rip; i++){
83             copy();
84         }
85
86         t1 = System.currentTimeMillis();
87
88         if (t1 - t0 <= tMin){
89             min = rip;
90         }else{
91             max = rip;
92         }
93     }
94
95     return max;
96 }
97
98
99
100 }
101
```

```

1 //
2 // Classe che genera numeri casuali,
3 // migliore del random di sistema
4 //
5 public class RandomGenerator {
6     //
7     // get(): restituisce un numero compreso tra 0 e 1
8
9     public double get()
10    {
11        //
12        // Costanti
13        //
14        final int a = 16087;
15        final int m = 2147483647;
16        final int q = 127773;
17        final int r = 2836;
18
19        //
20        // Variabili
21        //
22        double lo, hi, test;
23
24        hi = Math.ceil(seed / q);
25        lo = seed - q * hi;
26        test = a * lo - r * hi;
27        if (test < 0.0) {
28            seed = test + m;
29        } else {
30            seed = test;
31        } /* endif */
32        return seed / m;
33    }
34
35    //
36    // getSeed(): restituisce il valore corrente del seme
37    //
38    public double getSeed()
39    {
40        return seed;
41    }
42
43    //
44    // setSeed(s): imposta il valore del seme a s
45    //
46    public void setSeed(double s)
47    {
48        seed = s;
49    }
50
51    //
52    // costruttore della classe, genera un'istanza di RandomGenerator,
53    // fissando il seme iniziale a s.
54    //
55    public RandomGenerator(double s)
56    {
57        seed = s;
58    }
59
60    private double seed;
61
62    //
63    // esempio di uso della classe RandomGenerator,
64    // stampa 10 numeri casuali compresi tra 1 e 100
65    //
66    public static void main(String[] args)
67    {
68        long n, i;
69        //
70        // crea un istanza della classe RandomGenerator
71        //
72        RandomGenerator r = new RandomGenerator(123456789);
73
74        System.out.println("Il valore iniziale del seme e': " + r.getSeed());
75

```

```
76     for (i = 1; i <= 10; i++) {
77         n = Math.round(r.get() * 100);
78         System.out.println(n);
79     } /* endfor */
80 }
81
82 }
83
```