INGEGNERIA DEGLI ALGORITMI

PROGETTO N°2

A cura di

Simone Grillo	0258129
Michele Granatiero	0253496
Ivan Palmieri	0259244

10 Dicembre 2018

INDICE:

1. TRACCIA

Questo è un progetto sugli algoritmi di ordinamento e selezione. L'obbiettivo è quello di implementare una nuova versione dell'algoritmo select e di modificare l'algoritmo quickSort in cui la scelta del pivot, invece di avvenire in modo random, avviene tramite un algoritmo di selezione. Il nuovo algoritmo di selezione (sampleMedianSelect), basato su quello di Floyd e Rivest, sceglie il pivot su cui effettuare la partizione nel seguente modo:

• Sceglie un sottoinsieme V di m elementi in modo random

• Seleziona il mediano di V e lo usa come pivot

Come si può notare, questo algoritmo è una versione più generale dell'algoritmo deterministico select in cui il sottoinsieme V `e formato dai mediani delle quintuple. La scelta del parametro m `e lasciata allo studente.

Si richiede inoltre di confrontare sperimentalmente, al variare della dimensione della lista in input, il tempo di esecuzione delle varie varianti (con select randomizzata, deterministica e sampleMedianSelect) del nuovo algoritmo quickSort con:

• quickSort classico

• gli altri algoritmi di ordinamento visti e implementati a lezione

• il metodo sort() di Python

2. RICHIAMI TEORICI: L'ALGORITMO quickSort

L'algoritmo quickSort si basa sulla scelta randomica del pivot per effettuare la partizione. Scegliendo randomicamente il pivot si possono verificare 4 casi:

1. L'elemento scelto è il mediano
2. L'elemento scelto è il massimo
3. L'elemento scelto è il minimo
4. L'elemento scelto non è nessuno dei precedenti

Dall'analisi asintotica possiamo ridurre questi 4 casi a 2:

1. L'elemento scelto è il massimo o il minimo (o molto vicino ad essi)
2. L'elemento scelto è uno di tutti gli altri elementi

Proseguendo con l'analisi asintotica sappiamo che il caso peggiore si verificherà soltanto se il pivot scelto ricadrà nel primo caso per un tempo di esecuzione $O(n^2)$ altrimenti i tempi d'esecuzione del caso migliore o atteso saranno $O(nlogn)$. Pertanto il quickSort sfrutta la bassa probabilità di estrarre i valori di massimo o di minimo (o molto vicini ad essi)

3. L'ALGORITMO quickSortPrandom

L'algoritmo quickSortPrandom si propone di ridurre ulteriormente la probabilità di ricadere nel caso peggiore del quickSort, in favore del caso atteso.

Affinché ciò avvenga invece di estrarre un solo elemento, l'estrazione viene reiterata per m volte e ne si prende il mediano. Pertanto è cruciale la scelta del valore di m poiché più è grande, maggiore sarà la precisione nel trovare il mediano, tuttavia al crescere di m aumenta anche il costo computazionale per trovare il mediano stesso, portando al paradosso che per evitare di andare nel caso peggiore, si eseguano operazioni supplementari che facciano perdere il tempo guadagnato per la scelta del pivot ottimale.

Alla luce di ciò il quickSortPrandom è conveniente solo quando è valida la relazione

$$T(quickSortPrandom) < T(quickSort)_{caso\ peggiore}$$

Però sappiamo che entrambi gli algoritmi impiegano tempo $O(n)$ per la partizione, quindi la relazione diventa

$$T(mediano) < T(quickSort)_{caso\ peggiore} - T(partizione)$$

Quindi

$$T(mediano) < O(n^2) - O(n) \leq O(n^2)$$

3.1 LA SCELTA DEL VALORE DI m

Nella nostra implementazione del quickSortParandom m viene scelto in base all'ordine di grandezza della lista ordinare nella funzione recursiveQuickSort.

A questa funzione vengono passati come parametri la lista da ordinare, gli indici su cui poi verrà eseguita la partizione, e dei parametri booleani che determineranno quale variante quickSort eseguire:

Nel caso della variante Prandom, $m = log_{10}(distanza\ indici) + 1$: pertanto esisteranno al più 10^m elementi differenti, ora la probabilità di avere il caso peggiore per singola estrazione sarebbe $\frac{2}{10^m}$, quindi per $m = 1$ si avrebbe il 20% di probabilità di ricadere nel caso peggiore, ma per avere $m = 1$, la distanza tra gli indici deve essere 0, quindi la lista da ordinare deve essere o vuota o composta da un solo elemento. Quindi la probabilità massima che si può verificare di ricadere nel caso peggiore è del 2%, per questa scelta di m.

3.2 IL MEDIANO TRA GLI m: L'ALGORITMO sampleMedianSelect

Una volta stabilito il valore di m, esso viene dato come parametro insieme ai valori degli indici (left, right) alla funzione sampleMedianSelect. Quest'ultima crea una sottolista V di m tuple (valore, indice elemento lista).

Successivamente si valuta la dimensione di V: verrà usato il trivialSelect, se V è molto piccolo, per restituire l'indice del mediano rispetto alla lista salvandolo nella variabile mid, oppure V verrà ordinato tramite l'algoritmo di ordinamento mergeSort, salvando nella variabile mid il valore di V[$^m/_2 + 1$][1] (indice rispetto alla lista del mediano).

Una volta trovato l'indice del mediano, si scambiano gli elementi di lista[left] e lista[mid] e si esegue la funzione partition, sfruttando il meccanismo del quickSort deterministico che prende come pivot il primo elemento (quello di posizione left).

In generale il tempo di esecuzione della funzione sampleMedianSelect nel caso peggiore è

$$T(m) = O(mlogm)$$

Quindi

$$T(mediano) = O(mlogm)$$

Poiché $m < n$ allora:

$$T(mediano) = O(mlogm) < O(n^2)$$

4 RISULTATI SPERIMENTALI E GRAFICI

1. ALGORITMI DI TEMPISTICA $O(n^2)$

DIM. LISTA	SELECTIONSORT	INSERCTIONSORTDOWN	INSERCTIONSORTUP	BUBBLESORT
100	0.0	0.0	0.0	0.0
1000	0.0343668460	0.0249943971	0.0312421321	0.0140591621
10000	3.3891240119	2.7096731925	3.8020524740	1.4760579204
100000	338.96951104	273.02775278	378.59165534	148.99187885
200000	1389.0775099	1119.3102414	1557.7507861	617.09978142

Media dei tempi in secondi su test eseguiti per 10 volte

2. ALGORITMI DI TEMPISTICA $O(n^2)$ NEL CASO PEGGIORE, $O(nlogn)$ NEL CASO ATTESO

DIM. LISTA	QUICKSORTRAND	QUICKSORTDET.	QUICKSORTPRANDOM
100	0.0006248569	0.0003128385	0.0014059734
1000	0.0110970088	0.0056267932	0.0096992160
10000	0.1602294426	0.1050315010	0.1422504388
100000	6.2223044776	5.6564580560	7.1070392074
200000	22.761754738	21.699926849	24.311847531

Media dei tempi in secondi su test eseguiti per 100 volte

3. ALGORITMI DI TEMPISTICA $O(nlogn)$, $O(n)$ E IL sort() DI PYTHON

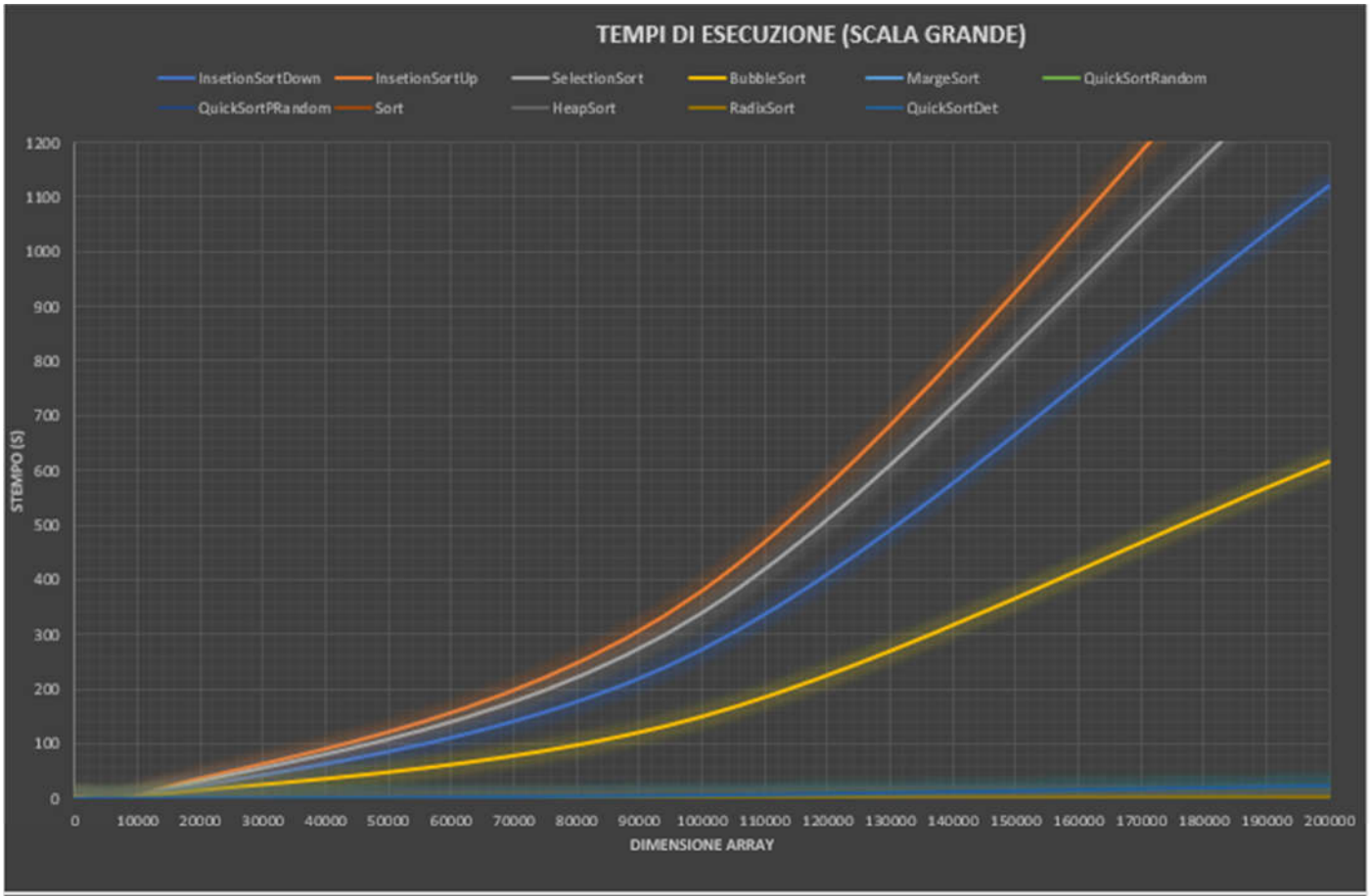
DIM. LISTA	MERGESORT	HEAPSORT	RADIXSORT	SORT PYTHON
100	0.0003127789	0.0006252169	0.0001561975	0.0
1000	0.0034394619	0.0107849510	0.0006263235	0.0
10000	0.0414312604	0.1463229641	0.0078172611	0.0
100000	0.4884238417	1.8339940380	0.0775596069	0.0006248688
200000	1.0230730082	3.9043273485	1.5638274942	0.0012559530

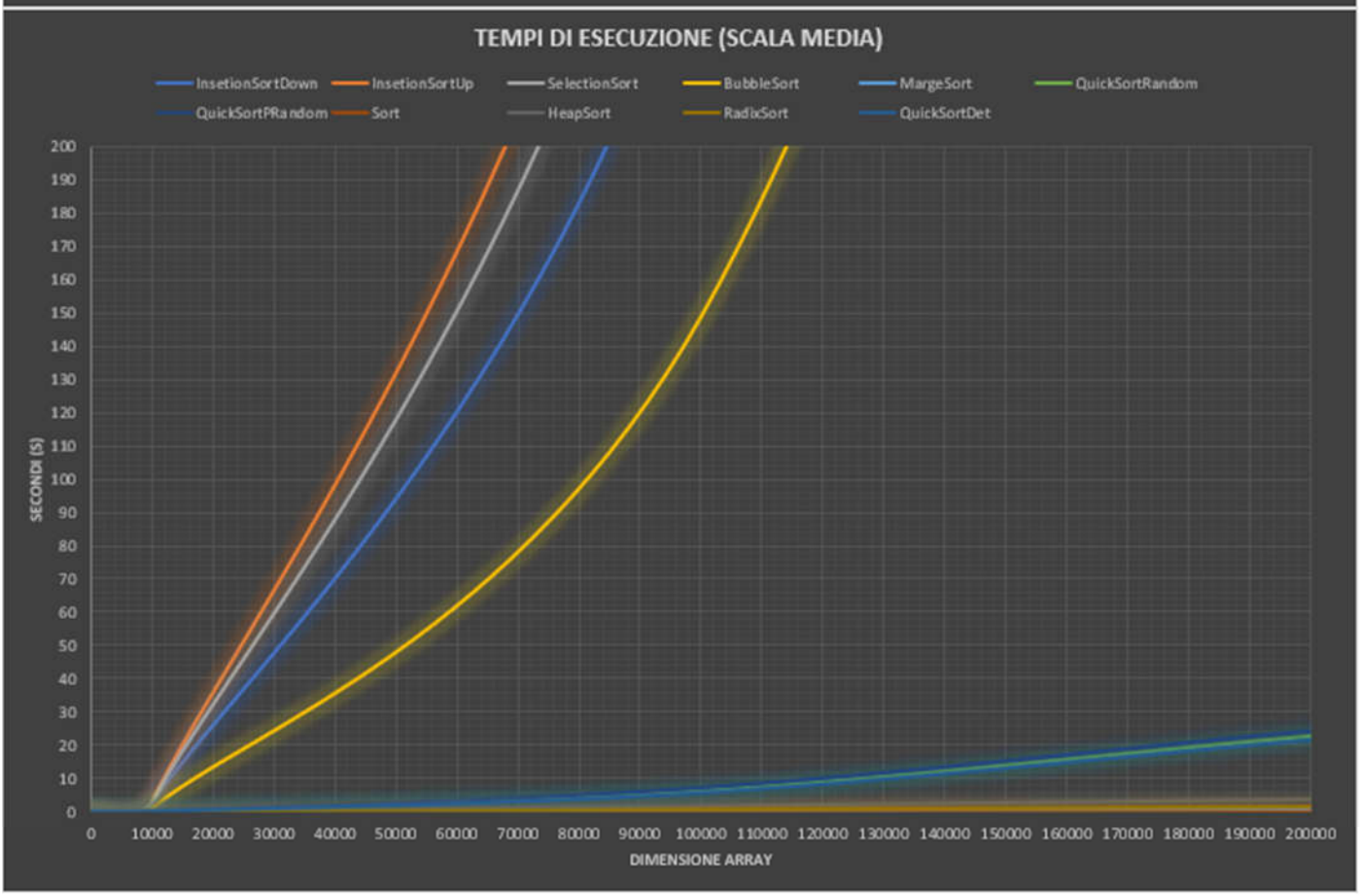
Media dei tempi in secondi su test eseguiti per 100 volte

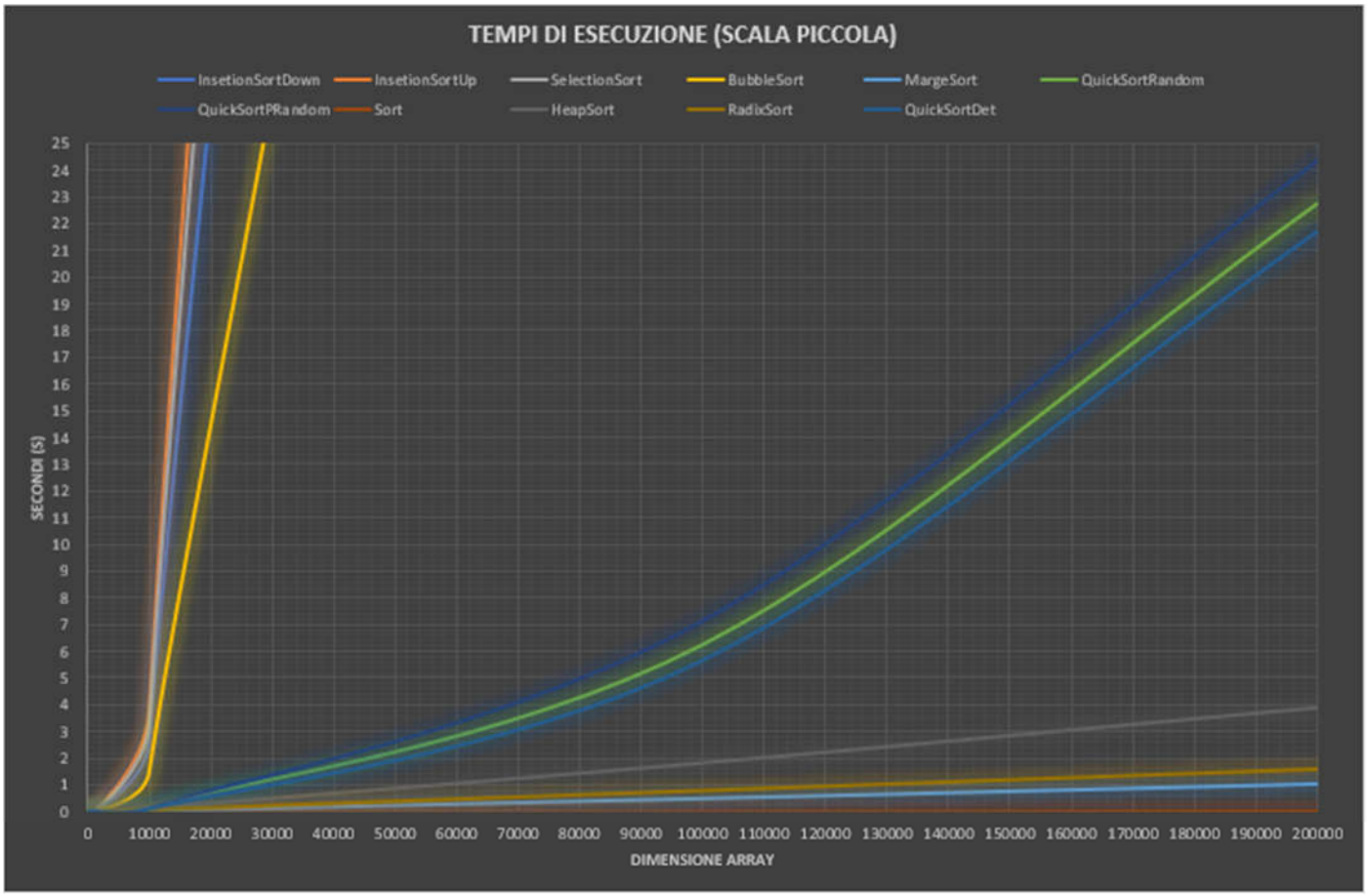
Dall'analisi dei tempi d'esecuzione si evince che come è ovvio che gli algoritmi del gruppo 1 abbiano dei tempi d'esecuzione peggiori rispetto a quelli del gruppo 2 e 3.

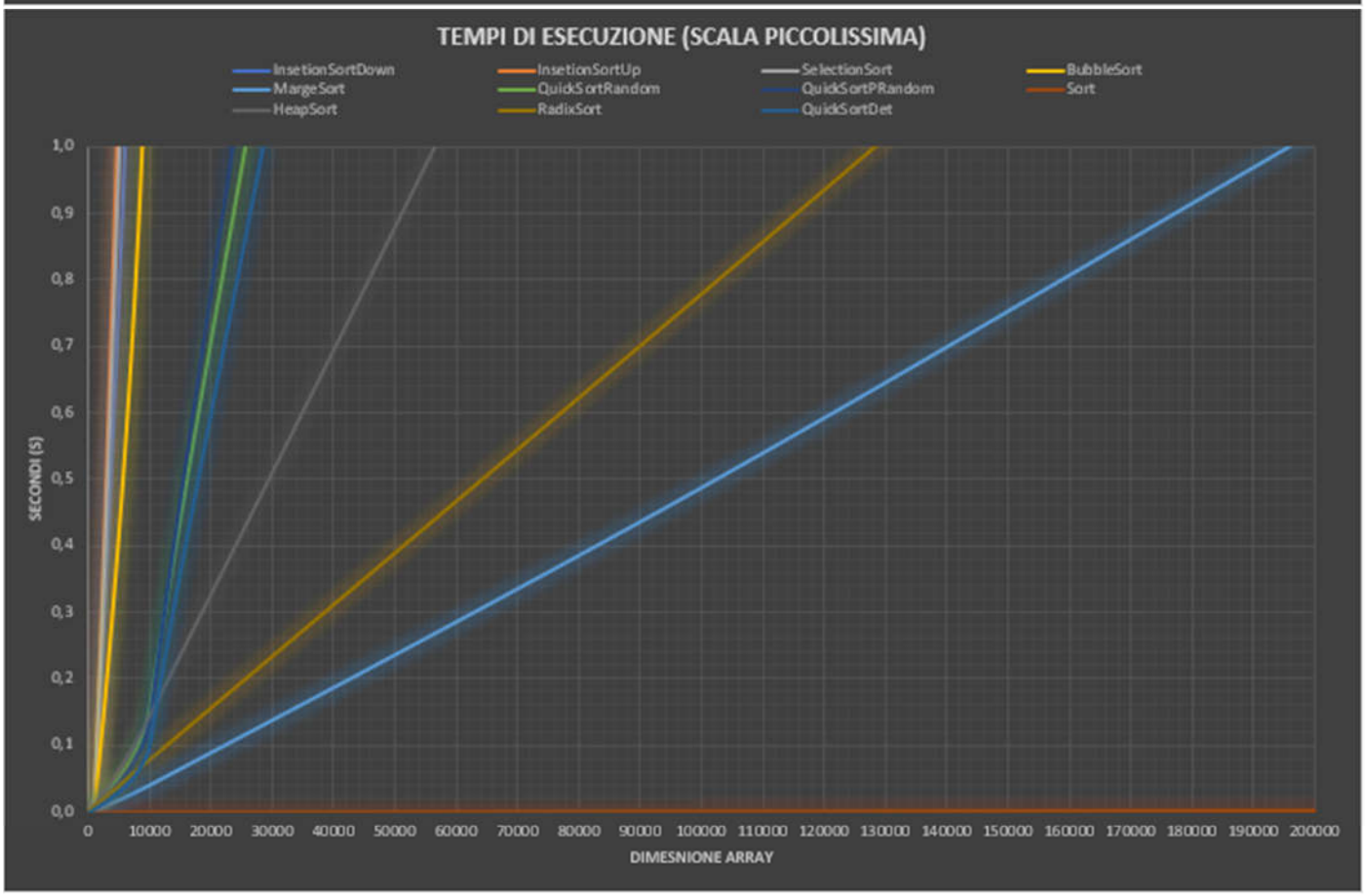
Analizzando i tempi d'esecuzione delle tre varianti del quickSort osserviamo come in media l'algoritmo quickSort deterministico sia più veloce delle altre due varianti, e che la variante Prandom sia la più lenta delle tre. Ciò avviene perché per effettuare le operazioni necessarie al ritrovamento del pivot "ottimale", vengono eseguiti più confronti rispetto alle altre due varianti, considerando il fatto che nel caso quickSort random sia comunque minima la possibilità di ricadere nel caso peggiore, in media l'andamento risulterà peggiore. Tuttavia per valori piccoli di input l'algoritmo Prandom ha comunque un andamento migliore rispetto alla sua variante randomica (per i valori compresi tra i 1000 e 10000).

Osservando l'andamento medio degli algoritmi del gruppo 2 rispetto agli algoritmi del gruppo 3, osserviamo che per input minori o uguali ai 200000 elementi conviene utilizzare gli algoritmi del 3° gruppo. Nello specifico osserviamo che l'algoritmo sort() di python risulta essere il migliore in assoluto, a seguire l'algoritmo radixSort fino ai 10000 elementi, che viene superato da qui in poi dal mergeSort. L'algoritmo heapSort risulta essere il più lento del 3° gruppo.


TEMPI DI ESECUZIONE (SCALA GRANDE)
InsetionSortDown
InsetionSortUp
SelectionSort
BubbleSort
MargeSort
QuickSortRandom
QuickSortPRandom
Sort
HeapSort
RadixSort
QuickSortDet
STEMPO (S)
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
110000
120000
130000
140000
150000
160000
170000
180000
190000
200000
DIMENSIONE ARRAY


TEMPI DI ESECUZIONE (SCALA MEDIA)
InsetionSortDown
InsetionSortUp
SelectionSort
BubbleSort
MargeSort
QuickSortRandom
QuickSortPRandom
Sort
HeapSort
RadixSort
QuickSortDet
SECONDI (S)
0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
150
160
170
180
190
200
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
110000
120000
130000
140000
150000
160000
170000
180000
190000
200000
DIMENSIONE ARRAY


TEMPI DI ESECUZIONE (SCALA PICCOLA)
InsetionSortDown
InsetionSortUp
SelectionSort
BubbleSort
MargeSort
QuickSortRandom
QuickSortPRandom
Sort
HeapSort
RadixSort
QuickSortDet
SECONDI (S)
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
110000
120000
130000
140000
150000
160000
170000
180000
190000
200000
DIMENSIONE ARRAY


TEMPI DI ESECUZIONE (SCALA PICCOLISSIMA)
InsetionSortDown
InsetionSortUp
SelectionSort
BubbleSort
MargeSort
QuickSortRandom
QuickSortPRandom
Sort
HeapSort
RadixSort
QuickSortDet
SECONDI (S)
0,0
0,1
0,2
0,3
0,4
0,5
0,6
0,7
0,8
0,9
1,0
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
110000
120000
130000
140000
150000
160000
170000
180000
190000
200000
DIMESNIONE ARRAY