**SCALETTA:**

* Copertina;
* Indice:
  + Strategia di Implementazione;
  + Raccolta ed Elaborazione Dati;
  + Approfondimenti;
* Strategia di Implementazione;
  + Idee scartate;
* Raccolta ed Elaborazione Dati;
  + Scrittura e uso di profiler.py;
  + Tabelle, grafici (**N.B:** riferire OS e possibilmente specifiche tecniche della macchina su cui sono stati effettuati i test) …
  + Studio di lenTuple, range / size …
  + Breve commento su ognuno dei risultati ottenuti
* Approfondimenti.
  + GitHub (?);
  + Argparse, random, plotter …
  + Qualsiasi cosa possa allungare il brodo …

**NOTE:**

* Ho usato font diversi in base a ciò che si scrive:
  + Codice: Courier New
  + Math:

Strategia di Implementazione

Per adempiere alle richieste presenti nella Traccia, è stata adottata la strategia di implementazione descritta di seguito:

* Implementazione di una variante dell’algoritmo QuickSort (**QS**), chiamata QuickSelectSort (**QSS**), che adopera un algoritmo di selezione per calcolare il pivot attorno a cui partizionare la lista di input;
* Implementazione di una variante dell’algoritmo Select (**S**), chiamata SampleMedianSelect (**SMS**), che estrae l’elemento desiderato k partizionando attorno a un pivot calcolato chiamando la funzione SampleMedian (**SM**).

Tali varianti sono state implementate nel file QSS\_module.py.

In fondo alla sezione è dedicato uno spazio per le strategie scartate.

Implementazione di QSS

A differenza del QS descritto nel modulo sorting.Sorting, QSS accetta come secondo parametro select {0, 1, 2}, il quale determina la variante di S che verrà invocata per calcolare il pivot attorno a cui verrà partizionata la lista da ordinare. Il vero e proprio lavoro di ordinamento è svolto da recursiveQuickSelectSort (**RQSS**), che rappresenta il nucleo ricorsivo. A quest’ultima sono passati altri due parametri, left e right, che indicano i confini della partizione su cui RQSS sta lavorando.

Una volta calcolato il pivot viene invocata sorting.partitionDet (**PD**), che effettua il lavoro di partizionamento e restituisce l’indice del pivot. Dunque, RQSS lavora ricorsivamente sulle partizioni ottenute, ordinando infine la lista di input.

Implementazione di SMS e SM

Diversamente da S (descritto nel modulo selection.Selection), SMS estrae l’elemento partizionando attorno a un pivot calcolato mediante l’invocazione di SM. A seconda se SMS lavora ricorsivamente su una delle due partizioni oppure restituisce l[indice(p)].

SM costruisce un sottoinsieme *V* di l partizionando la lista di input in tuple di lunghezza , estraendo un elemento a caso da ciascuna tupla e inserendolo in V. Se la condizione (con ) non si verifica ripete ricorsivamente le suddette operazioni, altrimenti si calcola il mediano di *V* invocando quickSelectRand (**QSR**) e lo restituisce.

Grazie al lavoro di partizionamento effettuato da SM, i valori estratti casualmente e inseriti in sono distribuiti in modo omogeneo su tutta la lista. Ciò aumenta la probabilità che il mediano estratto da corrisponda al mediano effettivo della lista, migliorando di conseguenza i tempi di esecuzione di SMS, poiché partiziona più velocemente.

È interessante notare come il parametro non sia un valore fissato, bensì in funzione di : gli effetti al variare di questi parametri sono discussi nella sezione di **Raccolta ed Elaborazione Dati**.

Strategie Scartate

Durante l’implementazione di SM, riguardo la scelta dell’algoritmo di selezione da utilizzare per estrarre il mediano da , è stato proposto trivialSelect (**TV**). Questo perché inizialmente la lunghezza delle tuple era stata fissata pari a 5, sulla falsariga di S. Studiando le variazioni dei tempi di esecuzione di SM al variare di , è stato scoperto che a un valore elevato di quest’ultimo è associato un miglioramento generale dei tempi di esecuzione di SM e SMS, oscurato da un ben più sensibile peggioramento dei tempi di TV, che presenta un tempo quadratico nel caso peggiore. Dunque, è stato ritenuto opportuno sostituire TV con una chiamata a QSR, che presenta un tempo nel caso peggiore.

Inoltre, mentre attualmente il parametro , ovvero la dimensione del sottoinsieme , può assumere valori tali che , inizialmente si era optato di definire , e dunque di costruire estraendo 3 elementi a caso dalla lista. L’idea è stata sostituita dal metodo delle partizioni, poiché quest’ultimo presenta una probabilità maggiore di restituire un valore il più vicino possibile al mediano della lista.

Raccolta ed Elaborazione Dati

In questa sezione ci occuperemo dell’analisi dei dati, la raccolta di essi tramite il file profiler.py, l’elaborazione attraverso grafici e tabelle ed infine il loro studio.

Implementazione profiler

Il codice all’interno dello script, essenzialmente, si riconduce alla chiamata degli algoritmi di ordinamento, ed attraverso l’importazione di moduli predisposti all’analisi tempistica delle funzioni chiamate quali cProfile e pstats, è possibile visionare la differenza del tempo d’esecuzione dei vari algoritmi.

* Il modulo cProfile fornisce un insieme di statistiche che descrive quanto spesso e per quanto tempo sono state eseguite varie parti del programma.
* Il modulo pstats interpreta ciò che cProfile ha ricavato dall’esecuzione del codice stesso.

L’utilizzo del modulo argparse è stato riservato invece per il passaggio in input, durante la chiamata del profiler, dei dati quali:

* La dimensione della lista;
* Range della funzione randint() per capire l’intervallo nel quale può andare a generare valori;
* Quale tra gli algoritmi di selezione eseguire in QSS;
* La possibilità di eseguire tutti gli altri algoritmi di ordinamento.

Grafici e Tabelle

Il supporto fisico su cui sono stati effettuati i test aveva le specifiche descritte in Figura 1.

Le misurazioni ottenute sono frutto di una media sviluppata in base al risultato di almeno 3 test.



Figura 1. Caratteristiche hardware della macchina su cui sono stati eseguiti gli esperimenti

Il primo grafico, in Figura 2, mostra il tempo di esecuzione devi vari algoritmi al variare della dimensione della lista in input.

A primo impatto è lampante la differenza tra gli algoritmi con tempo quadratico nel caso peggiore (selectionSort (**SS**), insertionSort (**IS**), BubbleSort (**BS**)) e gli altri algoritmi. Già con un’istanza in input di dimensione 20.000 si nota come essi siano poco efficienti. In Figura 3 riusciamo ad apprezzare meglio le differenze. La versione QSS che utilizza il select deterministico (**QSD**) impiega anche più di 10 secondi con liste la cui grandezza è dell’ordine di 105. L’HeapSort (**HS**) e la versione del QSS con SMS impiegano, invece, più di 4 secondi mentre i restanti, incluso il QSS con QSR, ne impiegano meno. L’algoritmo più efficiente è certamente il Sort di python.

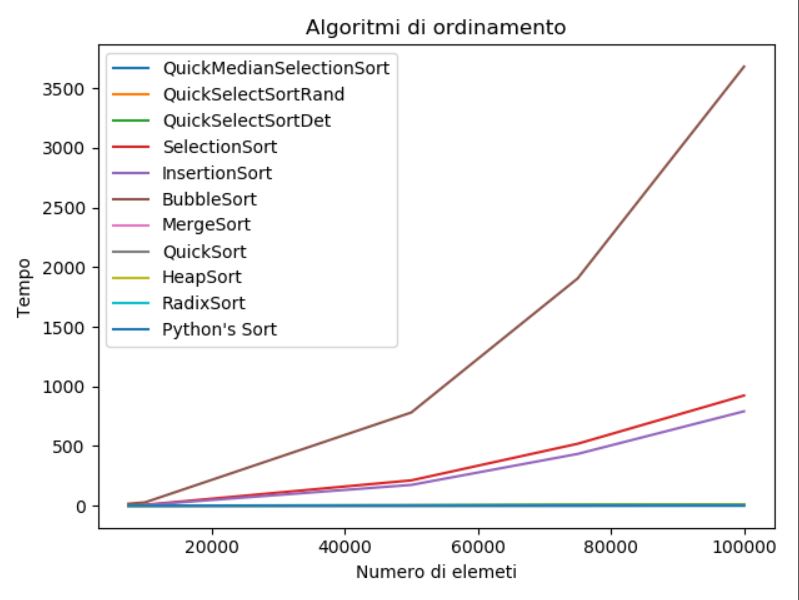


Figura Tempo di esecuzione degli algoritmi al variare della dimensione della lista

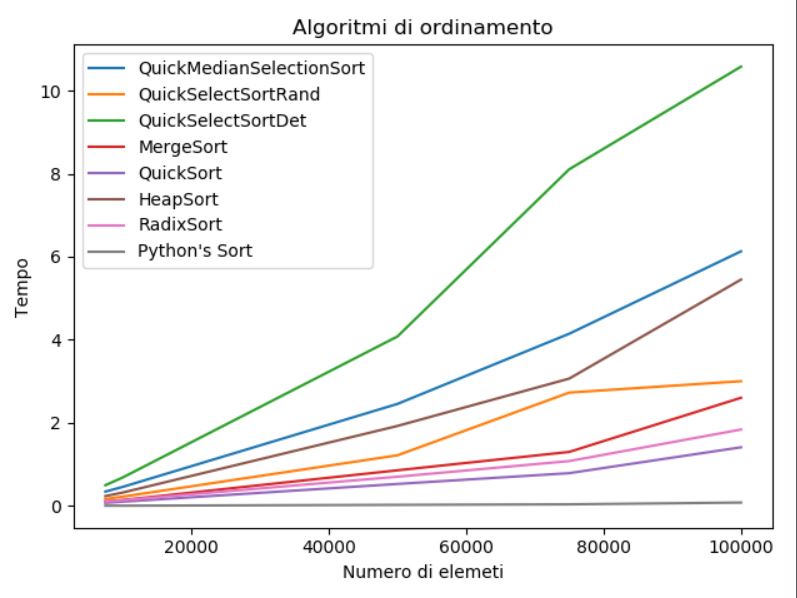


Figura . Zoom sulla Figura 2

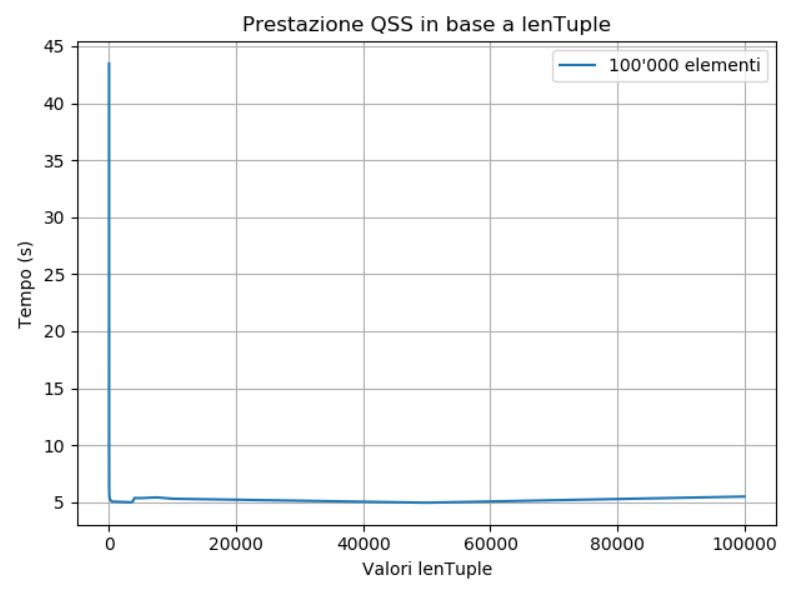
Nelle figure sottostanti sono presenti i risultati sperimentali dell’esecuzione di QSS al variare della dimensione di con la dimensione della lista in input pari a 100.000 elementi. Dal primo zoom sul grafico si capisce come una grandezza molto piccola delle tuple porta ad un aumento nel tempo di esecuzione sino ad arrivare a circa 43 secondi per una dimensione di corrispondente ad un singolo elemento. Nel secondo zoom invece si nota come per pari a 500 elementi il tempo di esecuzione diminuisce arrivando a circa 5 secondi, mentre nel terzo notiamo un calo al di sotto dei 5 secondi per ma abbiamo nuovamente un aumento del tempo per elementi. Nel quarto ed ultimo zoom è presente, per pari a 50.000 elementi (che in questo caso corrisponde proprio a ), una decrescita del tempo di esecuzione che si assesta a 4,951 secondi. Per un valore più alto di lenTuple abbiamo nuovamente un aumento nel tempo di esecuzione.

Figura . Tempo di esecuzione al variare di lenTuple

Nella scelta di quale valore utilizzare per l’implementazione di SMS si è scelto di utilizzare un valore di poiché inizialmente ritenuto un buon valore per l’ottimizzazione del tempo di esecuzione. I dati sperimentali dimostrano, invece, che i valori in un intorno sinistro di 3500 elementi e per un valore pari a si possono ottenere tempi migliori.

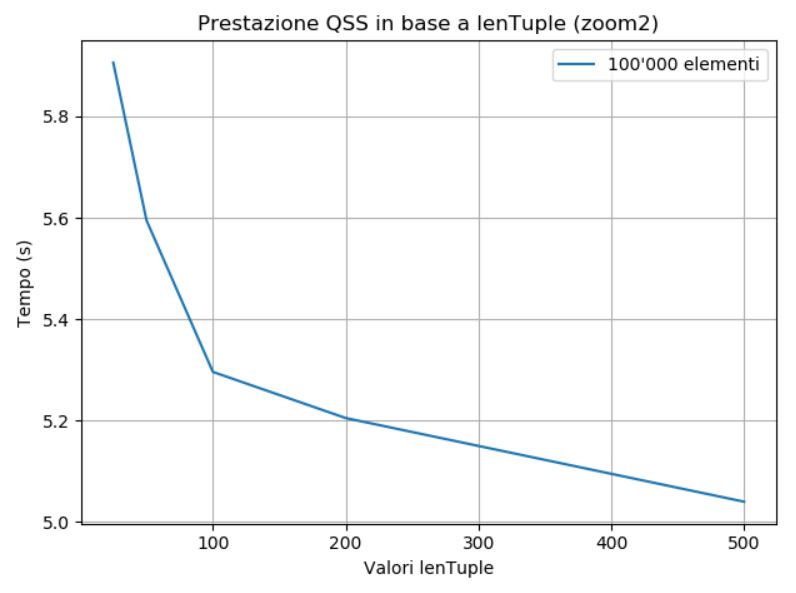
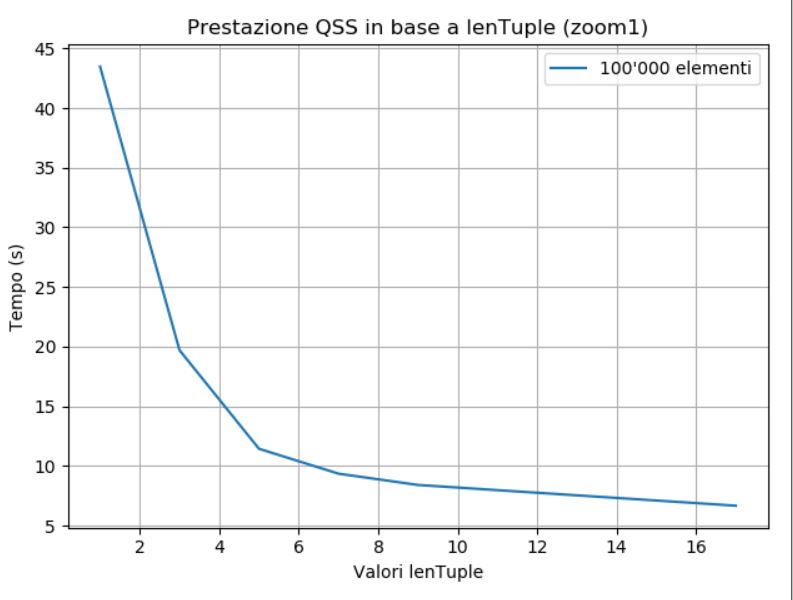


Figura . Zoom su valori di lenTuple compresi tra 1 e 17 Figura . Zoom su valori di lenTuple compresi tra 25 a 500

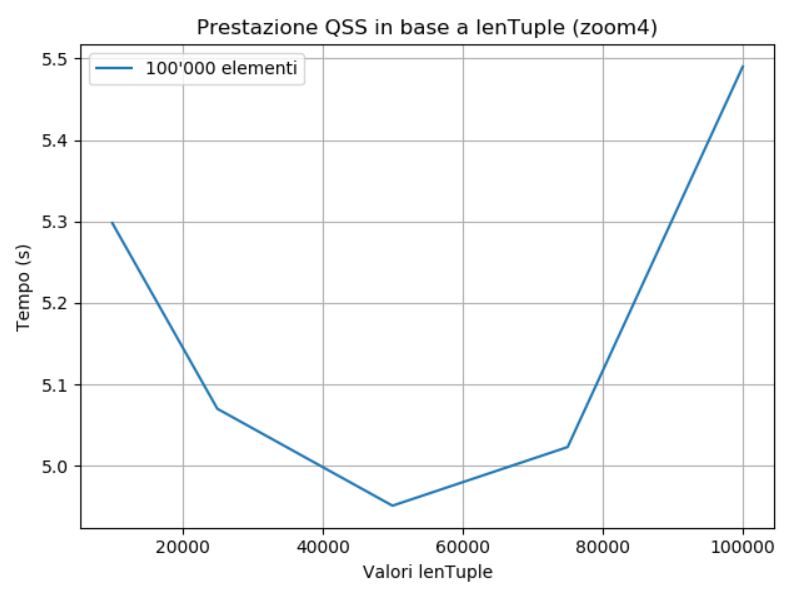
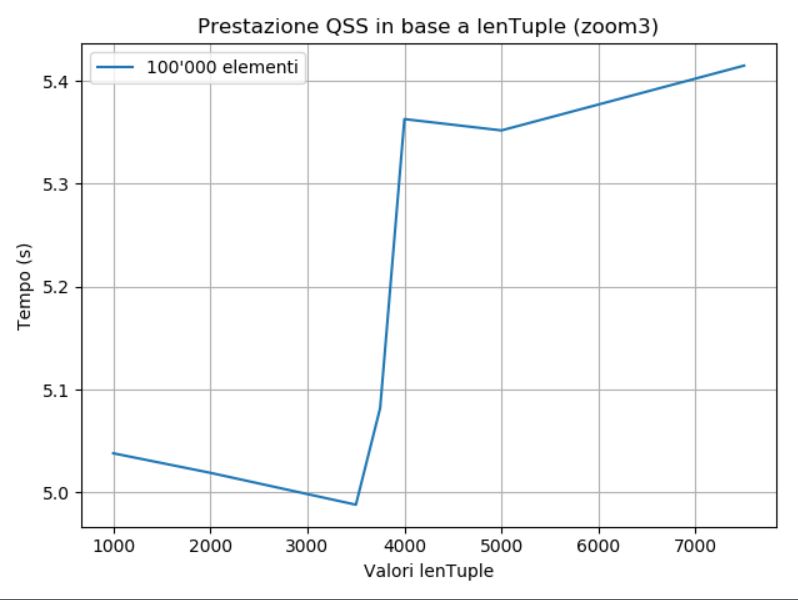


Figura . Zoom su valori di lenTuple compresi tra 1000 e 7500. Figura .Zoom su valori di lenTuple compresi tra 10000 e 100000

Un ulteriore fattore che può portare a casi sfavorevoli è il rapporto , dove range è l’intervallo di valori che possono essere rappresentati e size indica la dimensione della lista. Come mostrato in Figura 4 infatti, se questo rapporto produce un risultato eccessivamente basso, e quindi ci sarà un numero ridotto di esemplari unici all’interno della lista stessa, il tempo di esecuzione può superare i 14 secondi presentando così una prestazione inefficiente. Altrimenti, se , il tempo di esecuzione diminuirà al di sotto dei 6 secondi. Ciò dimostra che i casi peggiori si verificano con valori di prossimi allo zero.

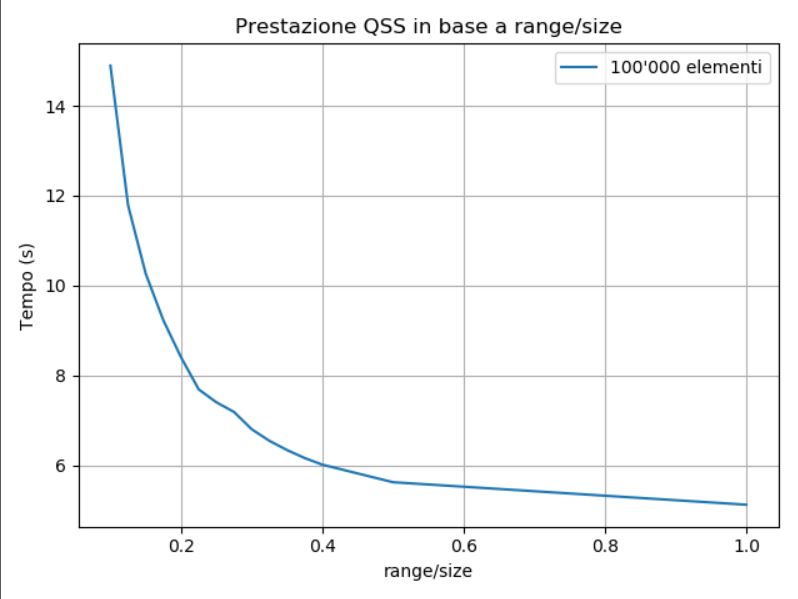


Figura .Tempo di esecuzione al variare del rapporto .