Cavaliere Alessandro Matricola: 0253983
Cavaliere Francesco Matricola: 0253942

1. Scelte implementative

Nella traccia ci è stato richiesto di implementare una nuova versione dell'algoritmo `select` e di implementarlo nell'algoritmo `quickSort` opportunamente modificato per scegliere il pivot tramite l'algoritmo di selezione `sampleMedianSelect` che, invece di prendere un pivot random, sceglie un sottoinsieme V di k elementi casuali, ne seleziona il mediano e lo usa come pivot.

L'algoritmo di selezione `sampleMedianSelect` è stato implementato in tale modo:

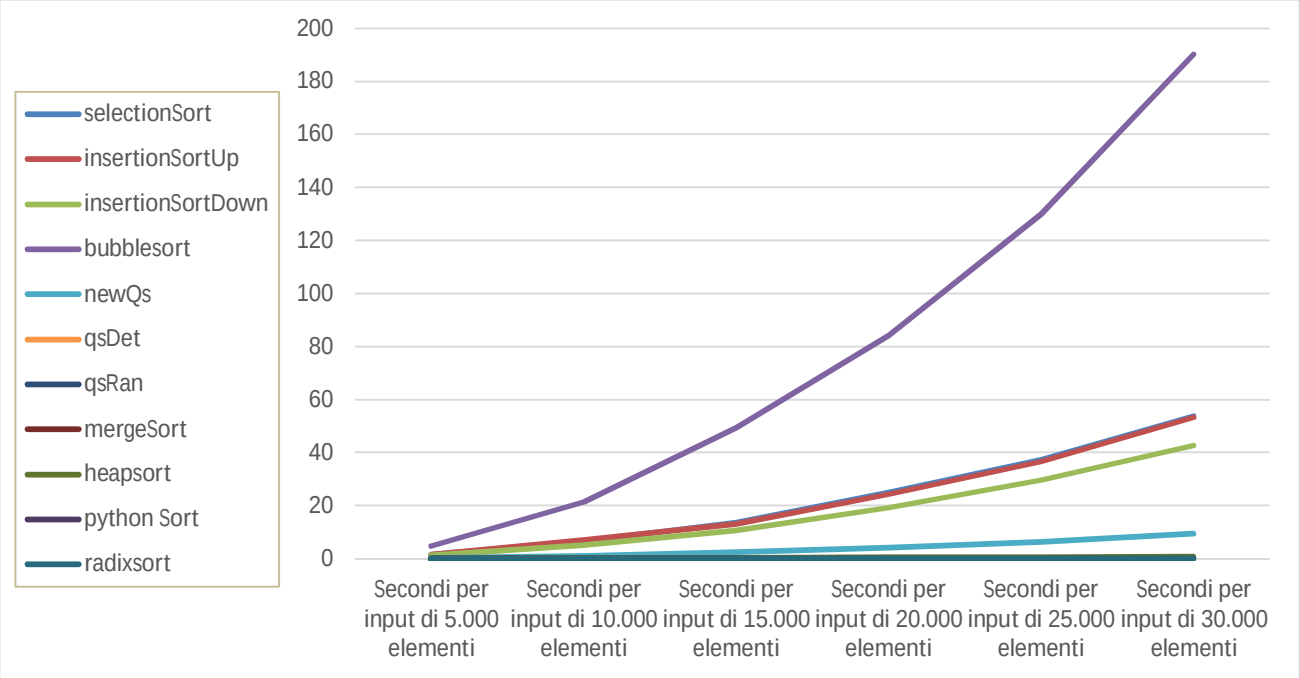
- `V = random.sample(a, 5) (riga 69)`genera un sottoinsieme di `k` elementi casuali nell'array, nel nostro caso k = 5, ma la scelta di k è libera, purché sia inferiore o uguale al numero di elementi presenti nell'array.
  `m = v[len(v)//2 ]` seleziona il valore del mediano nel sottoinsieme, e lo chiama m.
- Con la chiamata `mid = a.index(m) (riga 75)` possiamo trovare la posizione del mediano trovato in V, nell'array iniziale ed usarla nel nostro nuovo algoritmo `newQuickSort (come si vede nella riga 30)`
  Come il `quickSort` classico, anche in questo si effettuano più chiamate ricorsive. `(righe da 14 a 26)`
- Il restante codice funziona essenzialmente come il normale `quickSort`, ma riapplica l'algoritmo di selezione `sampleMedianSelect` nel caso in cui il mediano non possa più permetterci di ordinare l'array, trovandone un altro da utilizzare (righe da 33 a 37).

Per la realizzazione sono stati utilizzati:

- JetBrains PyCharm 2018.2.4
- Python 3.6
- Codici visti a lezione

Cavaliere Alessandro Matricola: 0253983
Cavaliere Francesco Matricola: 0253942

2. Tabelle e Grafici



Algoritmo	Secondi per input di 5.000 elementi	Secondi per input di 10.000 elementi	Secondi per input di 15.000 elementi
selectionSort	1,46406674385	6,52868986130	13,90265274048
insertionSortUp	1,37284779549	7,06596684456	13,07463860512
insertionSortDown	1,11453843117	5,06715989113	10,59255480766
bubblesort	4,82208061218	21,55928659439	49,56817078590
newQs	0,27926659584	1,07665681839	2,52144885063
qsDet	0,01894974709	0,04089021683	0,07377648354
qsRan	0,04986596107	0,11070394516	0,20446157455
mergeSort	0,03490686417	0,07682156563	0,15658235550
heapsort	0,08876323700	0,21028494835	0,33810639381
python Sort	0,00100755692	0,00199151039	0,00199484825
radixsort	0,01993584633	0,03986573219	0,06482625008

Algoritmo	Secondi per input di 20.000 elementi	Secondi per input di 25.000 elementi	Secondi per input di 30.000 elementi
selectionSort	25,01540994644	37,49480843544	53,93059372902
insertionSortUp	24,53067326546	36,84131908417	53,39568877220
insertionSortDown	19,37162184715	29,78865909576	42,71665573120
bubblesort	84,39922618866	129,87258863449	190,32057189941
newQs	4,08004307747	6,34757089614	9,40916657448
qsDet	0,08976006508	0,11073493957	0,13752555847
qsRan	0,22240471840	0,27928707800	0,34218311310
mergeSort	0,16257882118	0,20317554470	0,25410342216
heapsort	0,43834090233	0,56425976753	0,68815517426
python Sort	0,00298428535	0,00399231911	0,00498604774
radixsort	0,07881569862	0,10072922707	0,13410520554

3. Commento e conclusione

Come si può vedere dai risultati ottenuti, il nostro algoritmo `sampleMedianSelect` risulta essere più veloce di algoritmi più lenti, come: `selectionSort`, `insertionSortUp`, `insertionSortDown` e `bubbleSort`.

Possiamo inoltre notare che, con un basso numero di elementi, la differenza tra `sampleMedianSelect` e gli algoritmi d'ordinamento più veloci è ben poca, ma con l'aumentare del numero di elementi, il nostro algoritmo richiede un tempo sempre maggiore, che cresce molto più rapidamente rispetto agli altri.

Provando a modificare il numero k di elementi selezionati nel sottoinsieme v, abbiamo inoltre notato che, all'aumentare del valore k, il tempo richiesto aumenta a sua volta, analogamente, il tempo diminuisce per valori di k piccoli.

Dai valori sperimentati risulta che l'algoritmo `sort()` integrato in Python sia il più veloce da utilizzare a prescindere dalla grandezza dell'array.