Relazione progetto 14/11/2018 (Traccia 2)

Scelte implementative e funzionamento dell'algoritmo:

L'algoritmo presentato prende il nome di **quickSelectionSort** in quanto a tutti gli effetti variante di **quickSort**, dove il **pivot**, invece di essere preso in modo casuale, viene selezionato tramite le funzioni implementate nel corpo dell'algoritmo.

In primo luogo, avviene la creazione di un sottoinsieme **V** di **m** elementi, presi random dalla lista **l** tramite la funzione **randomIns** che, prendendo in input l e m intero, genera avvalendosi del metodo **random.randint()**.

Infine attraverso **sampleMedianSelect**, che prende in input l, left(=0), right(=len(l)), k(=len(l)/2) e m=3 nel nostro caso, risale al **mediano** (pivot, preso come valore, non indice) della lista l, basandosi sul mediano del sottoinsieme V selezionato in precedenza e agendo ricorsivamente, basandosi in buona parte sullo pseudocodice del**sampleSelect** presentato dal libro nel capitolo 5 riguardante la selezione.

Ottenuto il pivot, quickSelectionSort si comporta allo stesso modo di quickSort, tramite la tecnica del divide et impera, partizionando utilizzando **partition** (in questo caso secondo il mediano), gli elementi in due sequenze che vengono ordinate separatamente e ricorsivamente, per poi essere ricombinate a creare un nuovo array l in ordine ascendente.

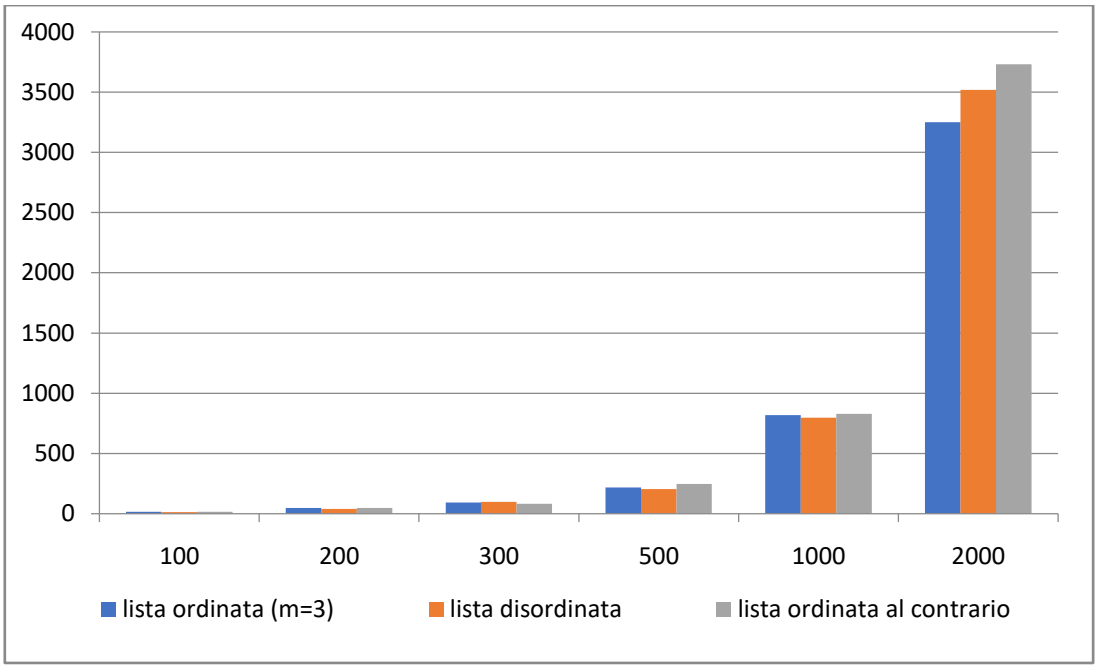
Risultati sperimentali:

I dati presenti in forma grafica e poi tabulare sono stati raccolti utilizzando la funzionalità di **profile** offerta da Pycharm 3.7, mostrante il tempo di esecuzione e il conteggio delle chiamate fatte di ogni funzione presente all'interno del codice.

Le caratteristiche della macchina utilizzata sono le seguenti:

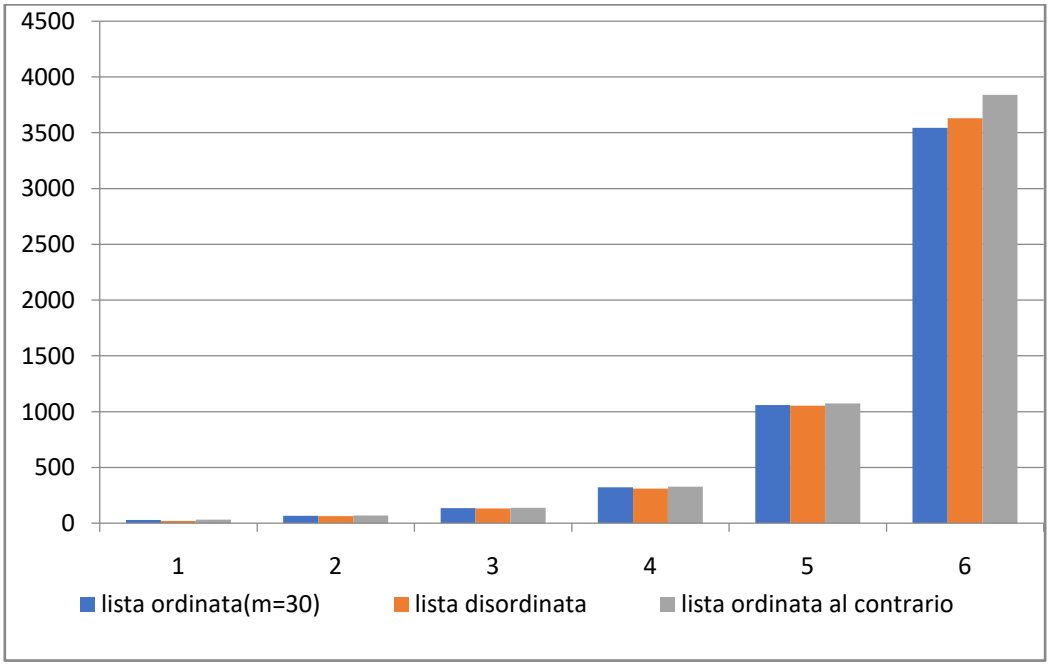
Processore:	Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz 2.80 GHz
Memoria installata (RAM):	16,0 GB (15,9 GB utilizzabile)

Segue quindi un confronto tra i tempi di esecuzione per liste di diversa lunghezza disposte in maniera ordinata ascendente, disordinata e ordinata discendente.



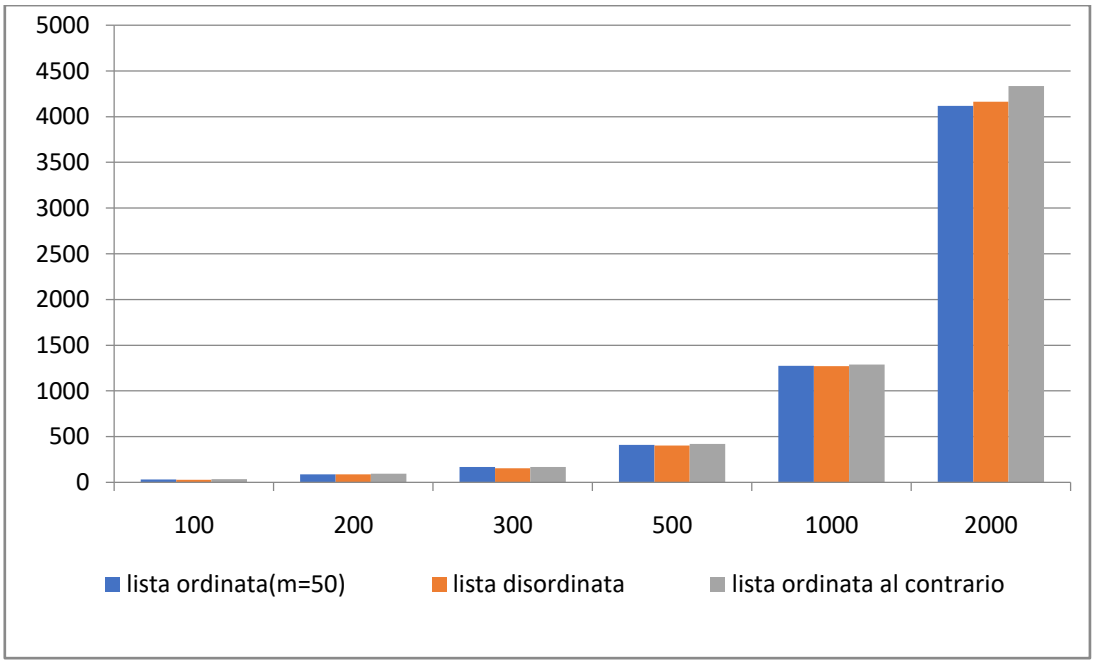
m = 3	INPUT (elementi)					
	100	**200**	**300**	**500**	**1000**	**2000**
Lista ordinata	15 (ms)	47 (ms)	92 (ms)	217 (ms)	819 (ms)	3251(ms)
Lista disordinata	14 (ms)	40 (ms)	99 (ms)	205 (ms)	798 (ms)	3520(ms)
Lista ordinata discendente	16 (ms)	49 (ms)	83 (ms)	248 (ms)	828 (ms)	3731(ms)

Provando a variare il parametro m (dimensione sottoinsieme V), si ha:



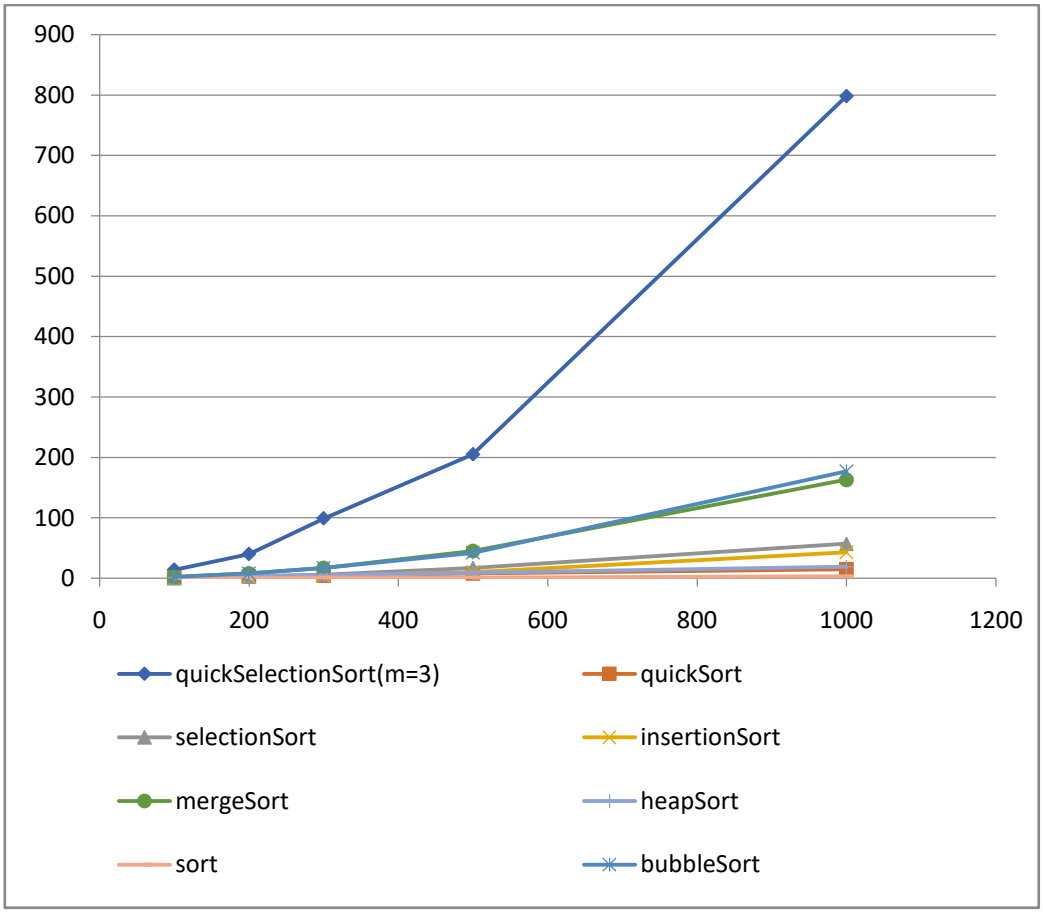
m = 30	**INPUT (elementi)**					
	100	**200**	**300**	**500**	**1000**	**2000**
Lista ordinata	28 (ms)	66 (ms)	135 (ms)	322 (ms)	1060(ms)	3544(ms)
Lista disordinata	19 (ms)	64 (ms)	133 (ms)	311 (ms)	1054(ms)	3631(ms)
Lista ordinata discendente	31 (ms)	69 (ms)	137 (ms)	327 (ms)	1073(ms)	3841(ms)

Per m = 50 invece:



m = 50	**INPUT (elementi)**					
	100	**200**	**300**	**500**	**1000**	**2000**
Lista ordinata	32 (ms)	89 (ms)	167 (ms)	410 (ms)	1274(ms)	4117(ms)
Lista disordinata	29 (ms)	88 (ms)	154 (ms)	404 (ms)	1272(ms)	4162(ms)
Lista ordinata discendente	36 (ms)	93 (ms)	167 (ms)	421 (ms)	1287(ms)	4336(ms)

Infine il confronto tra quickSelectionSort e gli algoritmi di ordinamento visti durante il corso e il metodo sort() di Python:



Commento dei risultati sperimentali ottenuti:

I risultati ottenuti confermano ciò che la logica accennava. L'algoritmo quickSelectionSort richiede molto più tempo rispetto ai classici visti a lezione già solamente per un input di 1000 elementi, mentre il metodo sort() di Python è decisamente il più veloce se confrontato a tutti gli altri. I tempi migliori per quickSelectionSort si ottengono scegliendo il parametro **m** quanto più piccolo possibile, risparmiando quindi tempo di esecuzione nella funzione di creazione del sottoinsieme e nella scelta del mediano di V, operando meno confronti. L'algoritmo richiede meno tempo quando gli si da in input una lista già in ordine ascendente, mentre ne richiede più nel caso dove la lista sia in ordine discendente che in quello dove è completamente disordinata.