Progetto 2: Relazione

•Introduzione

Tutti i test sono stati eseguiti su una macchina con processore Intel® Core™ i7-8550U e 8 GB di memoria RAM, con kernel Linux stable (4.19.9), tramite l'ambiente di sviluppo PyCharm di JetBrains.

Il codice è stato caricato su GitHub al link https://github.com/Lorenzoval/midterm-algoritmi/

La commit history è stata mantenuta il più pulita possibile, in modo tale da poter osservare con chiarezza ogni singola modifica effettuata. Il codice riguardante il progetto si trova nella cartella project, fatta eccezione per una singola modifica al codice di partition (commit 7574da0).

I grafici sono stati realizzati con Excel, sull'asse delle ascisse c'è la dimensione della lista in input, su quello delle ordinate c'è il tempo di esecuzione dell'algoritmo in secondi.

•Stesura del codice e primi test pratici

Per prima cosa è stata implementata una diversa versione della funzione sample della libreria random (project/modules/Sample.py). La funzione è stata testata (project/modules/demoSample.py) insieme a quella della libreria e, essendo risultata più lenta, non è stata sfruttata per la realizzazione dell'algoritmo `sampleMedianSelect`. L'esperimento, tuttavia, permette di notare che l'estrazione di un insieme random di m elementi (così come l'estrazione del mediano) richiede tempo maggiore al crescere di m, per questo è stato scelto un m "piccolo".

`quickSort` è stato modificato nei commit b265788 e 4637981. Il pivot viene scelto tramite un algoritmo di selezione ricevuto come parametro e la partition viene effettuata su un elemento della lista da ordinare sfruttando la funzione `partitionDet` importata da selection/Selection.py. Gli algoritmi di selezione (O(n) nel caso medio) estraggono il mediano da una copia della lista da partizionare, la relazione di ricorrenza del nuovo `quickSort` sarà quindi:

$T(n) = O(n) + O(n) + 2 * T(n / 2) = 2 * T(n / 2) + O(n)$

Quest'ultima, per il Teorema Master, ammette come soluzione $\Theta(n * \log(n))$.

Per le ragioni sopraccitate, `sampleMedianSelect` è stato codificato supponendo di ricevere in input un m abbastanza piccolo da considerare `trivialSelect` una valida opzione per l'estrazione del mediano del sottoinsieme V. Quando l'algoritmo ricorre su una lista di dimensione minore o uguale a quella di m, questo ne restituisce direttamente il risultato di `trivialSelect`. La funzione `quickSelectRand` è stata la base per l'implementazione di `sampleMedianSelect`.

Tra gli m "piccoli" è stato scelto quello che rappresentava il miglior compromesso tra la possibilità di evitare il caso peggiore e il dover estrarre il mediano da un insieme di elementi troppo grande. In project/selection/demoM.py sono stati eseguiti diversi test, facendo una media dei tempi di esecuzione al variare di m. Dai primi test il miglior m sembrava essere 2, ma il mediano di due elementi è il più piccolo tra loro: la funzione non differiva quindi abbastanza da

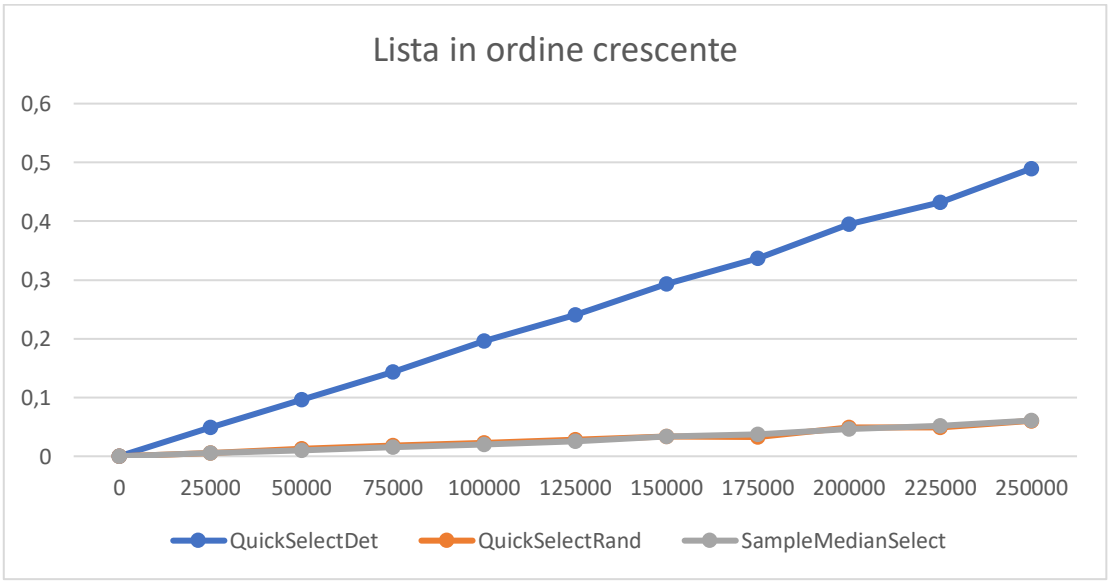
`quickSelectRand`. È stato quindi scelto il valore maggiore di 2 che garantiva tempi di esecuzione (sebbene questi variassero di pochissimo) mediamente migliori, ovvero m = 4.

•Realizzazione dei grafici e commento dei risultati

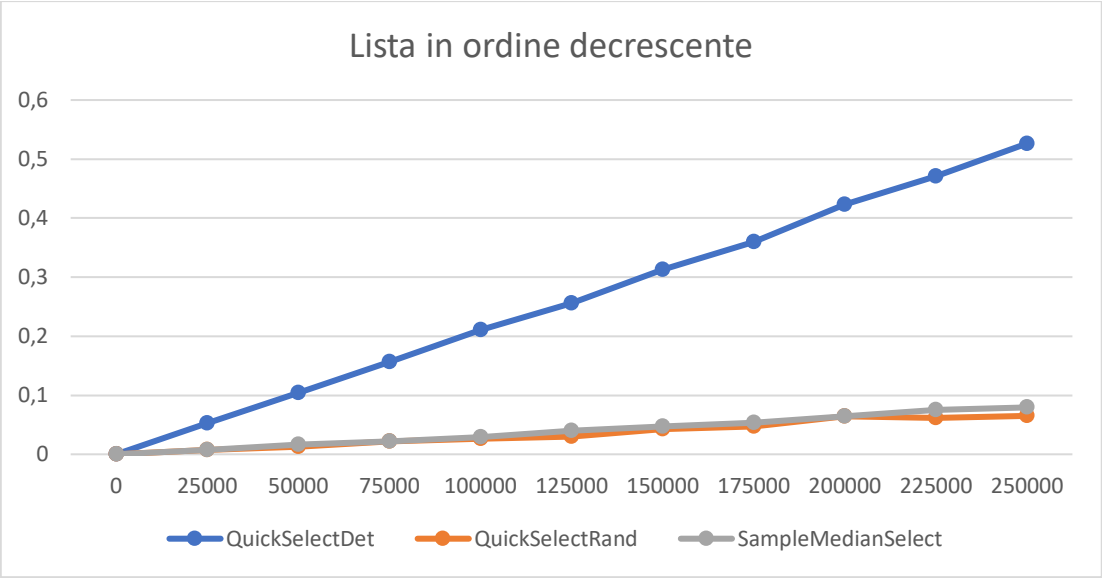
`sampleMedianSelect` è stato confrontato con `quickSelectRand` e `quickSelectDet` nel file project/selection/demoSelection.py. Per ogni algoritmo è stata graficata la media dei tempi di esecuzione in 20 misurazioni differenti, avendo come input liste di dimensione crescente (fino a 250000 elementi) ordinate in modo casuale, crescente o decrescente. I risultati sono i seguenti:

Random	0	25000	50000	75000	100000	125000	150000	175000	200000	225000	250000
QSD	0	0,053554	0,110883	0,16697	0,222026	0,279629	0,335554	0,394858	0,455696	0,513987	0,569294
QSR	0	0,010371	0,01699	0,02841	0,037971	0,04447	0,055806	0,074066	0,079928	0,079676	0,089095
SMS	0	0,009273	0,016932	0,028429	0,038126	0,049703	0,061007	0,069302	0,084434	0,099588	0,104413

Crescente	0	25000	50000	75000	100000	125000	150000	175000	200000	225000	250000
QSD	0	0,048794	0,096396	0,143842	0,195941	0,240968	0,293104	0,336447	0,394433	0,432262	0,489516
QSR	0	0,005686	0,012754	0,017847	0,022901	0,027913	0,033818	0,033067	0,049389	0,048894	0,060349
SMS	0	0,005579	0,00998	0,015219	0,020133	0,025929	0,033403	0,037239	0,045994	0,051379	0,060863



Decrescente	0	25000	50000	75000	100000	125000	150000	175000	200000	225000	250000
QSD	0	0,05247	0,103988	0,156422	0,210632	0,255866	0,312882	0,359838	0,423384	0,470483	0,526237
QSR	0	0,007341	0,012817	0,021907	0,026607	0,030271	0,042408	0,047253	0,064463	0,061671	0,065023
SMS	0	0,007476	0,016093	0,022112	0,029259	0,040284	0,047059	0,053409	0,064137	0,075099	0,079405

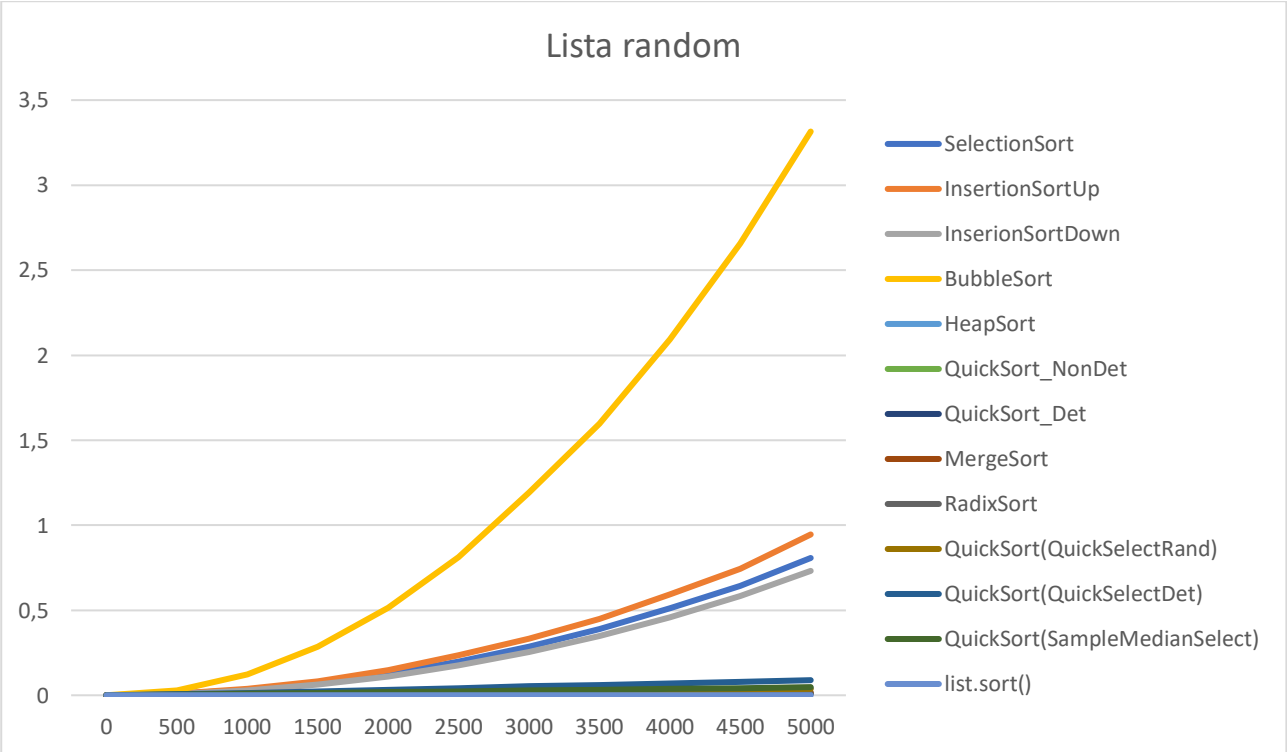


Come si può notare, `sampleMedianSelect` ha tempi di esecuzione molto vicini a quelli di `quickSelectRand`. Mentre `sampleMedianSelect` permette di imbattersi nel caso peggiore con probabilità più bassa, `quickSelectRand` non deve calcolare il mediano di un sottoinsieme random ad ogni passo ricorsivo. Interessanti sono invece i tempi di esecuzione di `quickSelectDet`, che dimostrano che l'algoritmo è di elevata importanza teorica (garantisce un tempo di esecuzione O(n)), ma di difficile applicazione pratica.

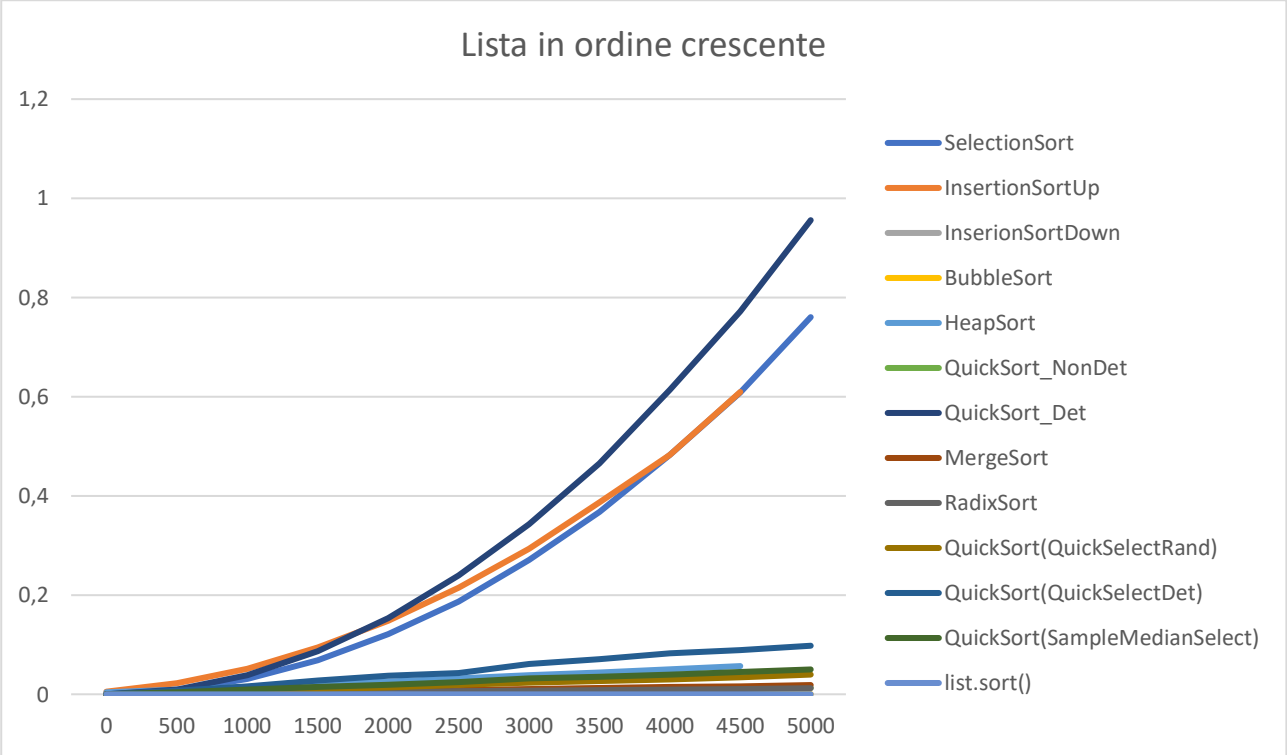
La nuova versione di `quickSort` è stata confrontata con tutti gli algoritmi di ordinamento visti a lezione (project/sorting/demoSorting.py) , usando `sampleMedianSelect`, `quickSelectRand` e `quickSelectDet` per determinare il pivot.

Anche in questo caso, i tempi presenti sui grafici sono quelli risultanti dalla media su 20 misurazioni. Gli algoritmi quadratici sono stati testati solo con liste di massimo 5000 elementi, ordinate in modo casuale, crescente o decrescente. Gli algoritmi più veloci, invece, sono stati confrontati anche su liste con massimo 100000 elementi in ordinamento casuale. I risultati sono i seguenti:

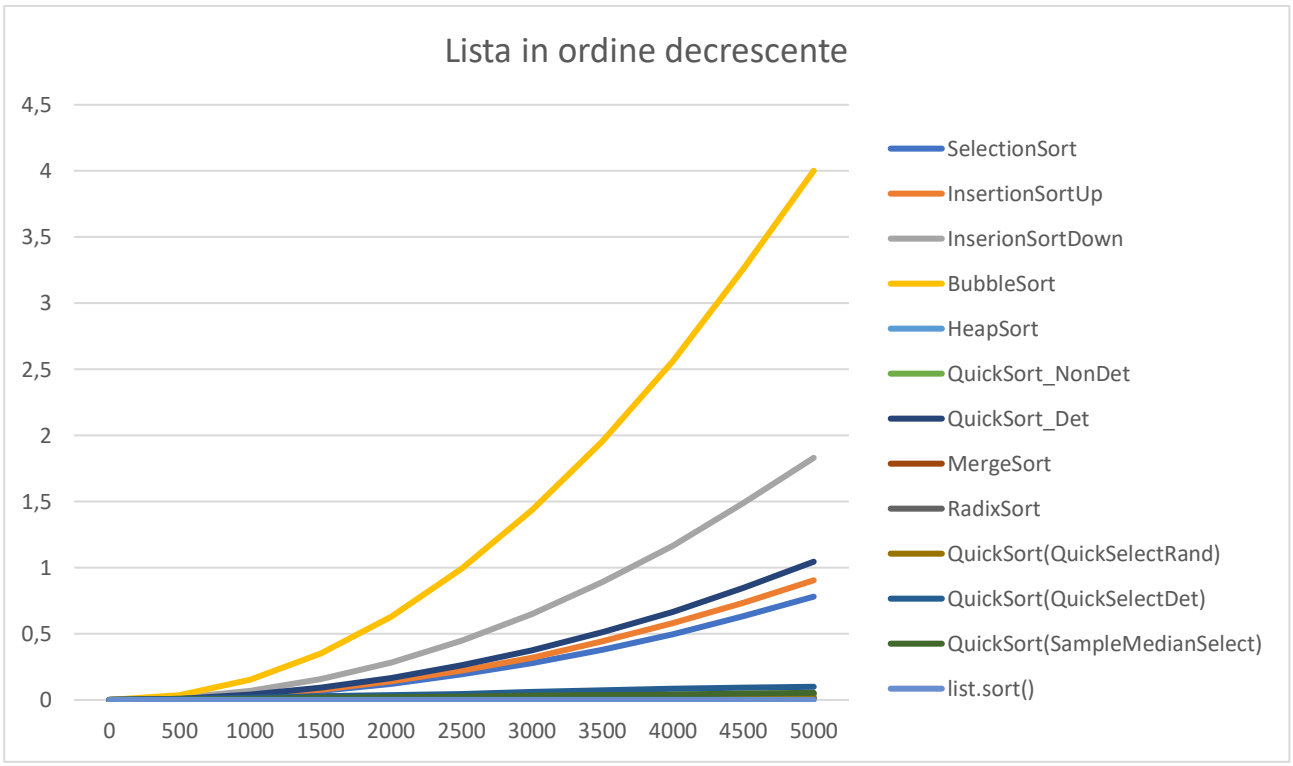
Random	0	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
Selection	0	0,00825	0,032055	0,071994	0,127123	0,197767	0,286229	0,390515	0,512147	0,64304	0,807446
InsUp	0	0,00872	0,036439	0,08253	0,147163	0,234311	0,334386	0,450457	0,592711	0,744339	0,945797
InsDown	0	0,006356	0,02713	0,06189	0,111258	0,17563	0,255289	0,349251	0,457407	0,582723	0,731468
Bubble	0	0,028175	0,122682	0,285152	0,51529	0,812663	1,191142	1,597537	2,091375	2,655749	3,315318
Heap	0	0,003067	0,006899	0,010806	0,014953	0,01913	0,023483	0,027475	0,032594	0,036496	0,041549
Quick_ND	0	0,001009	0,002151	0,003442	0,004566	0,0058	0,007267	0,008181	0,009498	0,010726	0,012137
Quick_D	0	0,000651	0,001464	0,002326	0,003326	0,00403	0,00491	0,005741	0,006868	0,007691	0,008492
Merge	0	0,001473	0,00333	0,004768	0,006528	0,008329	0,010366	0,01194	0,013707	0,015747	0,017896
Radix100	0	0,001121	0,002177	0,003026	0,003999	0,004817	0,005711	0,006749	0,007601	0,008567	0,00959
QS_QSR	0	0,003123	0,00689	0,010456	0,014151	0,017864	0,021897	0,025385	0,029375	0,033596	0,037797
QS_QSD	0	0,006164	0,014294	0,023421	0,031661	0,039955	0,052125	0,060979	0,069999	0,078517	0,088872
QS_SMS	0	0,004197	0,00886	0,013574	0,017995	0,022861	0,028513	0,032955	0,037904	0,042514	0,04796
Built-in	0	3,26E-05	7,34E-05	0,000116	0,000155	0,000201	0,000243	0,000288	0,000346	0,0004	0,000462



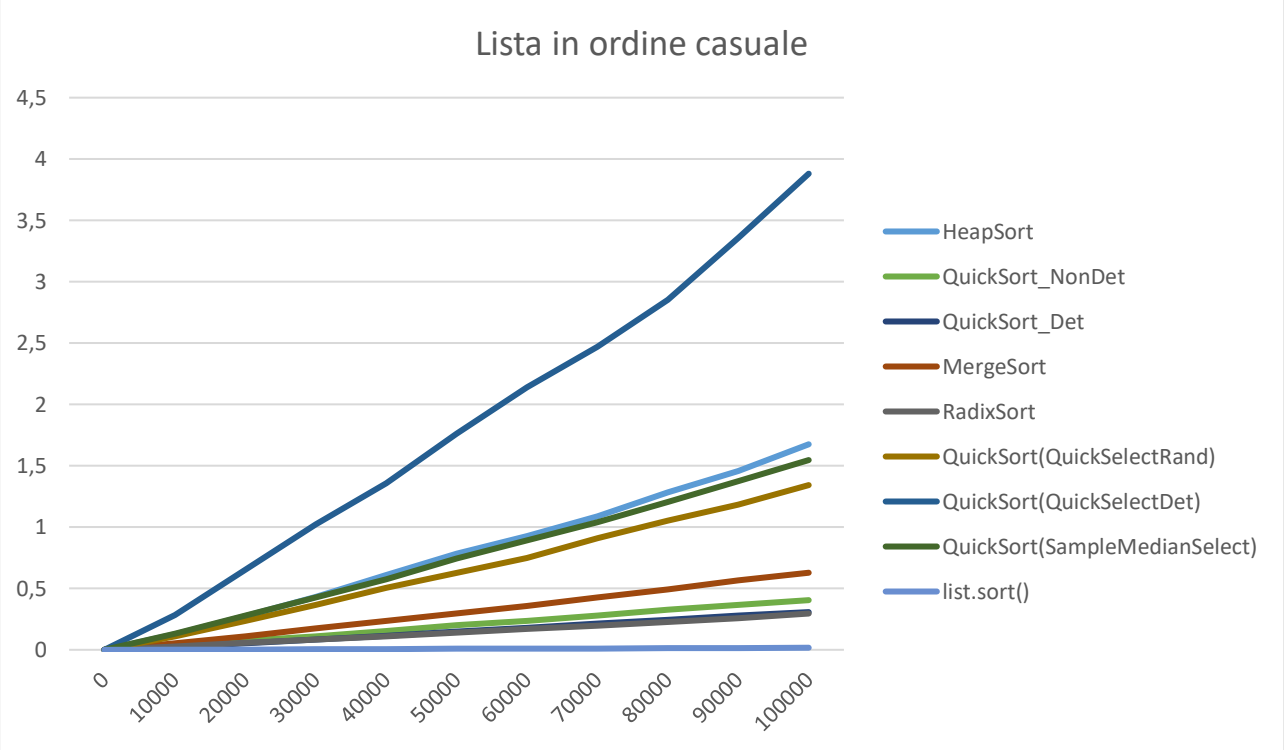
Crescente	0	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
Selection	0	0,007695	0,030629	0,068196	0,121362	0,187474	0,271311	0,367185	0,482779	0,609016	0,76043
InsUp	0	0,00532	0,022705	0,051408	0,094902	0,148175	0,214387	0,292885	0,387209	0,482326	0,609565
InsDown	0	0,000261	0,000545	0,000825	0,001115	0,001386	0,001686	0,001956	0,002259	0,002527	0,002834
Bubble	0	4,22E-05	8,84E-05	0,00013	0,000183	0,000235	0,000272	0,000333	0,000379	0,000404	0,000474
Heap	0	0,004076	0,009065	0,01455	0,019993	0,025936	0,032018	0,03799	0,044222	0,050143	0,056873
Quick_ND	0	0,001067	0,002317	0,003566	0,004909	0,006281	0,007508	0,008861	0,010234	0,011639	0,013079
Quick_D	0	0,009834	0,038996	0,086608	0,153857	0,239641	0,34291	0,465867	0,614201	0,772388	0,955941
Merge	0	0,001457	0,003166	0,004968	0,006789	0,008725	0,010658	0,01248	0,014392	0,016467	0,018622
Radix100	0	0,001386	0,002589	0,003809	0,004989	0,00618	0,007458	0,008626	0,009855	0,010983	0,012293
QS_QSR	0	0,003155	0,006728	0,011064	0,014231	0,018943	0,023438	0,026441	0,029785	0,034654	0,03974
QS_QSD	0	0,006922	0,016204	0,027308	0,036916	0,043138	0,061436	0,071295	0,083031	0,089192	0,0979
QS_SMS	0	0,004261	0,009117	0,015328	0,018862	0,024183	0,031931	0,035532	0,039253	0,044865	0,050092
Built-in	0	2,23E-06	3,86E-06	5,73E-06	7,65E-06	9,49E-06	1,16E-05	1,36E-05	1,57E-05	1,78E-05	1,98E-05

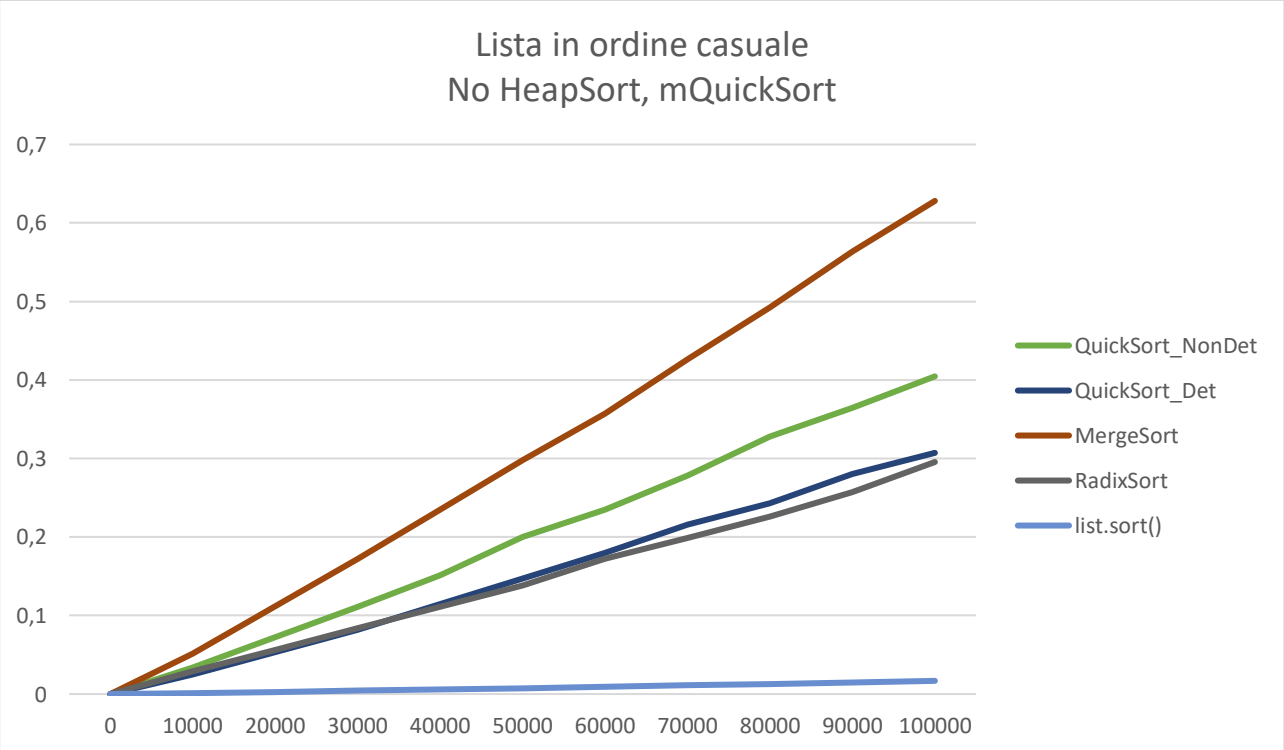


Decr	0	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
Selection	0	0,007754	0,031588	0,069345	0,123059	0,192143	0,279799	0,377879	0,495913	0,6333	0,781091
InsUp	0	0,007616	0,033888	0,077348	0,140239	0,221812	0,320776	0,442167	0,579525	0,73462	0,9046
InsDown	0	0,015976	0,069165	0,157858	0,284187	0,447476	0,650095	0,889826	1,165172	1,487402	1,830674
Bubble	0	0,035476	0,15283	0,351507	0,629089	0,99148	1,434139	1,955361	2,561217	3,25904	4,001337
Heap	0	0,00345	0,007845	0,012643	0,01754	0,022752	0,028007	0,033638	0,038849	0,044433	0,050194
Quick_ND	0	0,001097	0,002355	0,003681	0,004969	0,006358	0,00768	0,009103	0,010401	0,011915	0,01337
Quick_D	0	0,010098	0,041663	0,094072	0,167551	0,261505	0,376728	0,513079	0,667209	0,84629	1,045162
Merge	0	0,001447	0,003148	0,004908	0,006726	0,008589	0,010449	0,012253	0,014209	0,016178	0,018202
Radix100	0	0,001372	0,002581	0,00377	0,004949	0,006126	0,007313	0,008483	0,009678	0,010981	0,01216
QS_QSR	0	0,003241	0,006928	0,011391	0,01446	0,019211	0,023792	0,027175	0,030681	0,035375	0,040336
QS_QSD	0	0,007023	0,016439	0,027861	0,037517	0,043805	0,062208	0,072626	0,084943	0,091201	0,099434
QS_SMS	0	0,004601	0,009628	0,016389	0,020139	0,025879	0,034292	0,038009	0,041706	0,047938	0,053649
Built-in	0	2,47E-06	4,04E-06	5,95E-06	8,02E-06	1,00E-05	1,22E-05	1,44E-05	1,67E-05	1,90E-05	2,11E-05



Fast_Rand	0	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
Heap	0	0,125936	0,273152	0,430899	0,610564	0,782534	0,925603	1,087206	1,282641	1,458244	1,674763
Quick_ND	0	0,033985	0,072732	0,111346	0,1512	0,200104	0,234791	0,278369	0,327702	0,364643	0,404615
Quick_D	0	0,024643	0,053653	0,081744	0,114527	0,147453	0,179651	0,215803	0,243294	0,280649	0,307193
Merge	0	0,051371	0,111521	0,172504	0,234756	0,297884	0,357121	0,426418	0,492071	0,563887	0,628107
Radix400	0	0,028795	0,056311	0,083932	0,11127	0,138324	0,172291	0,198535	0,226183	0,257048	0,295554
QS_QSR	0	0,109823	0,235726	0,36747	0,505101	0,626045	0,750346	0,909704	1,054378	1,184063	1,342394
QS_QSD	0	0,283243	0,654205	1,020727	1,355821	1,759544	2,137896	2,470981	2,853352	3,357638	3,880303
QS_SMS	0	0,131826	0,277352	0,426421	0,57514	0,743284	0,89041	1,038359	1,203512	1,37592	1,546181
Built-in	0	0,00125	0,002692	0,004297	0,005859	0,007537	0,009316	0,01107	0,012923	0,014887	0,016802





Dai dati raccolti si evince che, nel caso di liste ordinate in modo crescente, il `quickSort` deterministico è il meno efficiente, poiché sceglie sempre il minimo della lista come pivot, cadendo nel suo caso peggiore. In questo scenario `bubbleSort` e `insertionSortDown` si trovano nel loro caso migliore. La versione modificata di `quickSort`, se si lavora con liste ordinate, risulta più efficiente della versione deterministica, ma meno efficiente della versione randomizzata, la quale non deve estrarre il mediano ad ogni passo. La versione deterministica d'altra parte risulta migliore se si lavora con liste in ordine casuale. Le considerazioni finali sono espresse sul metodo built-in di Python: list.sort(). Questo algoritmo ha come caso migliore un input ordinato (ordine di 10^{-5} s per liste di 5000 elementi) e mantiene un'efficienza elevata anche su liste random, tanto da superare tutti gli altri algoritmi implementati.