|  |
| --- |
| **HC – HasCycle** |
| Progetto 2 - Algoritmi di verifica della presenza di un ciclo in un grafo non orientato connesso. |
|  |
| Si presenta l’implementazione dell’algoritmo hasCycle nelle sue versioni DFS e UF, insieme a un algoritmo che genera grafi non orientati connessi e un decoratore con ruolo di profiler. |
|  |
| **Mihai Jianu, Daniele La Prova, Lorenzo Mei** |
| **[IA1819]** |
|  |

Indice

* Copertina; …………………………………………………………………………..………………1
* Indice; …………………………………………………………………………………………………2
  + Strategia di Implementazione;…………………………………………………3
  + Implementazione di QSS;…………………………………………………………3
  + Implementazione di SM e SMS;……………………………………………….3
  + Strategie Scartate;……………………………………………………………………4
* Raccolta ed elaborazione dati;……………………………………………………….....5
  + Implementazione Profiler;………………………………………………………..5
  + Grafici e Tabelle;……………………………………………………………………….5
* Approfondimenti;
* Cenni storici;……………………………………………………………………………14

Strategia di Implementazione

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

* Implementazione dell’algoritmo gGenerator (**GG**), che genera grafi non orientati connessi (a)ciclici rappresentati come matrice di adiacienza;
* Implementazione dell’algoritmo hasCycle (**HC**), che verifica la presenza di un ciclo nel tipo di grafi sopracitati sfruttando la visita DFS (**HC\_DFS**) o la struttura dati Union-Find (**HC\_UF**);
* Implementazione del decoratore @profiler (**P**), che regsitra dati prestazionali delle funzioni su cui è applicato e li annota in un file di testo.

Implementazione di GG

Riguardo la rappresentazione dei grafi è stato deciso di utilizzare come struttura dati le matrici di adiacenza, in quanto il tempo di esecuzione richiesto per l’operazione utilizzata più frequentemente da HC, ovvero la verifica dell’adiacenza tra due nodi, richiede tempo Questo vantaggio è ottenuto al prezzo di un impiego di memoria dell’ordine di , che limita la dimensione dei grafi intorno ai 10000 nodi.

**ATTENZIONE:** un tentativo di generazione di un grafo con un numero di nodi può comportare seri rallentamendi della macchina, fino al freeze totale. Per tale ragione, il valore di è limitato dalla costante .

L’algoritmo dietro al funzionamento di GG consiste in pochi, semplici passi:

In altre parole, GG prende in input il numero di nodi , il range di rappresentazione dei valori dei nodi e il numero di cicli richiesto . Per volte, esegue un pop di un nodo tail casuale da una composta inizialmente da nodi. Inoltre, sceglie un altro nodo casuale dalla (senza eseguire il pop) e inserisce nel grafo gli archi e . In questo modo si ottiene un grafo non orientato connesso con nodi e archi. Inoltre, aggiunge archi al grafo tra nodi non adiacienti tra loro,  inserendo dunque cicli.

Il tempo di esecuzione previsto è se il grafo richiesto è aciclico.

Se è richiesta l’aggiunta di cicli, il tempo di esecuzione sale a , con nel caso peggiore.

L’implementazione di GG è scritta nel modulo GG\_module.py.

Implementazione di HCDFS

Il file HCDFS\_module.py contiene l’implementazione dell’algoritmo HCDFS e le relative funzioni che consentono il suo funzionamento.

Partendo dalla funzione haCycleDFS(), essa assume in input il parametro G, ovvero il grafo, ed altri dati opzionali quali:

* debug, booleano che permette di visualizzare informazioni in più per il debugging;
* showProfile, per la visualizzazione a schermo delle informazioni che vengono scritte dal Decorator all’interno del file (Vedi sezione D\_module).

Al suo interno viene creata un’istanza di CustomGAM (**CGAM**), ovvero una sottoclasse della classe GraphAdjacencyMatrix. Si è scelto di implementare questa sottoclasse per non alterare la struttura del metodo dfs() presente in Graph.py per la rilevazione di cicli nei grafi.

In seguito viene chiamato il metodo dfsDetectedCycle() presente in CGAM, che prende in input i parametri rootId (nodo di partenza della visita DFS) e debug. Esso controllerà attraverso la visita DFS se è presente o meno un ciclo in G.

La visita DFS viene implementata attraverso uno stack e vengono aggiunti al suo interno tutti i nodi adiacenti al vertice preso in considerazione che non sono ancora stati esplorati. Questo vuol dire che se un nodo (A) è presente all’interno dello stack almeno 2 volte, allora esso è stato “visto” da due nodi diversi ma a lui adiacenti (B, C adiacenti ad A). Perché questo dovrebbe implicare la presenza di un ciclo? Perché B e C sono connessi tra loro o attraverso un arco (Figura 1.1) oppure attraverso un altro nodo (Figura 1.2). Infatti se ci siamo fermati sul nodo B, abbiamo “visto” tutti i suoi nodi adiacenti, abbiamo messo A nello stack ed in seguito siamo passati al nodo C, allora esso deve necessariamente essere connesso a B, direttamente o attraverso un altro vertice, in quanto il nodo successivo da esaminare è scelto attraverso un pop dalla pila stessa.

Quindi viene effettuato un controllo sullo stack ed attraverso un contatore si esamina quante volte è presente un dato nodo al suo interno.

Questa scelta nella risoluzione del problema evita molti inconvenienti, uno dei tanti ad esempio è la confusione di un arco per un ciclo (essendo il grafo non orientato), in particolare dell’arco incidente al nodo in cui ci troviamo ed il vertice esaminato precedentemente.

Il tempo di esecuzione è analogo a quello di una vista DFS su grafi implementati come matrici di adiacienza, ovvero .

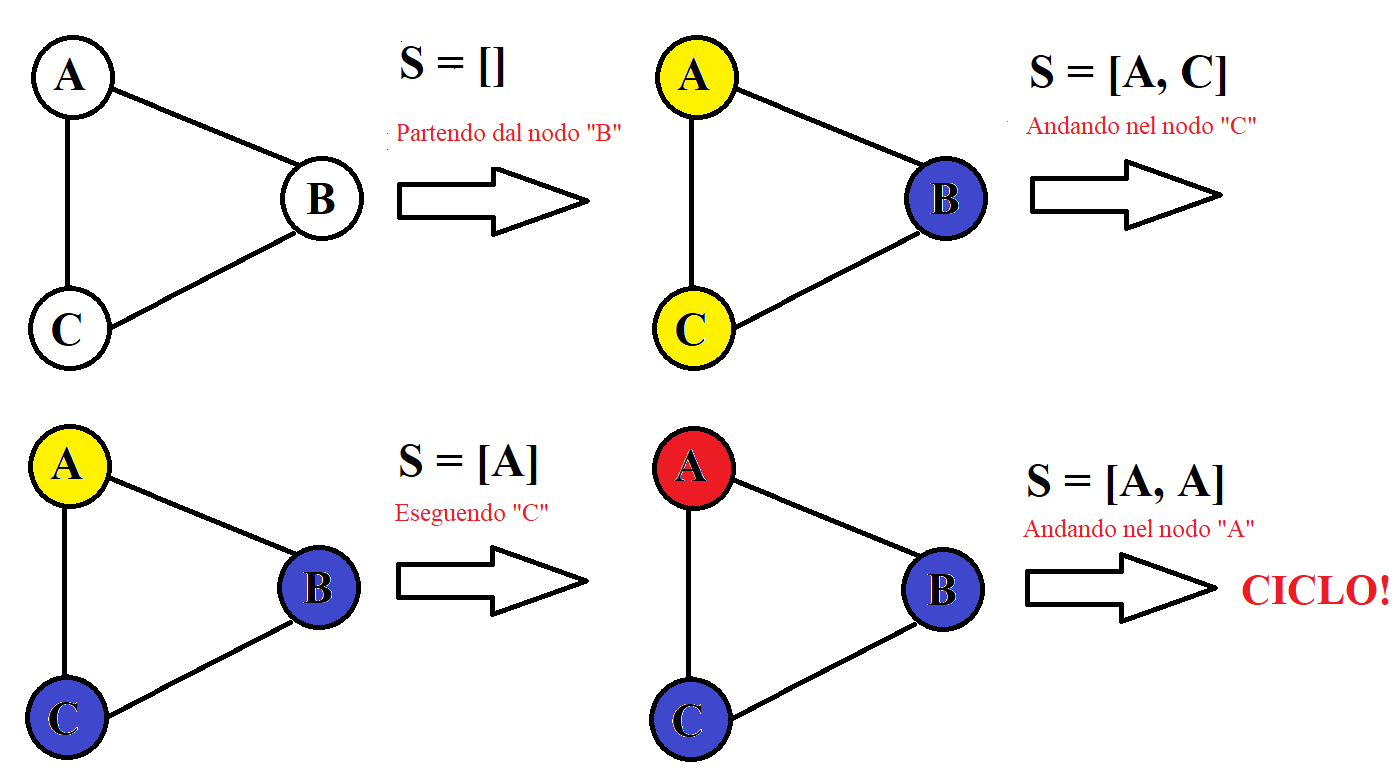


Figura 1. Caso di ciclo in cui B e C sono connessi tramite un arco

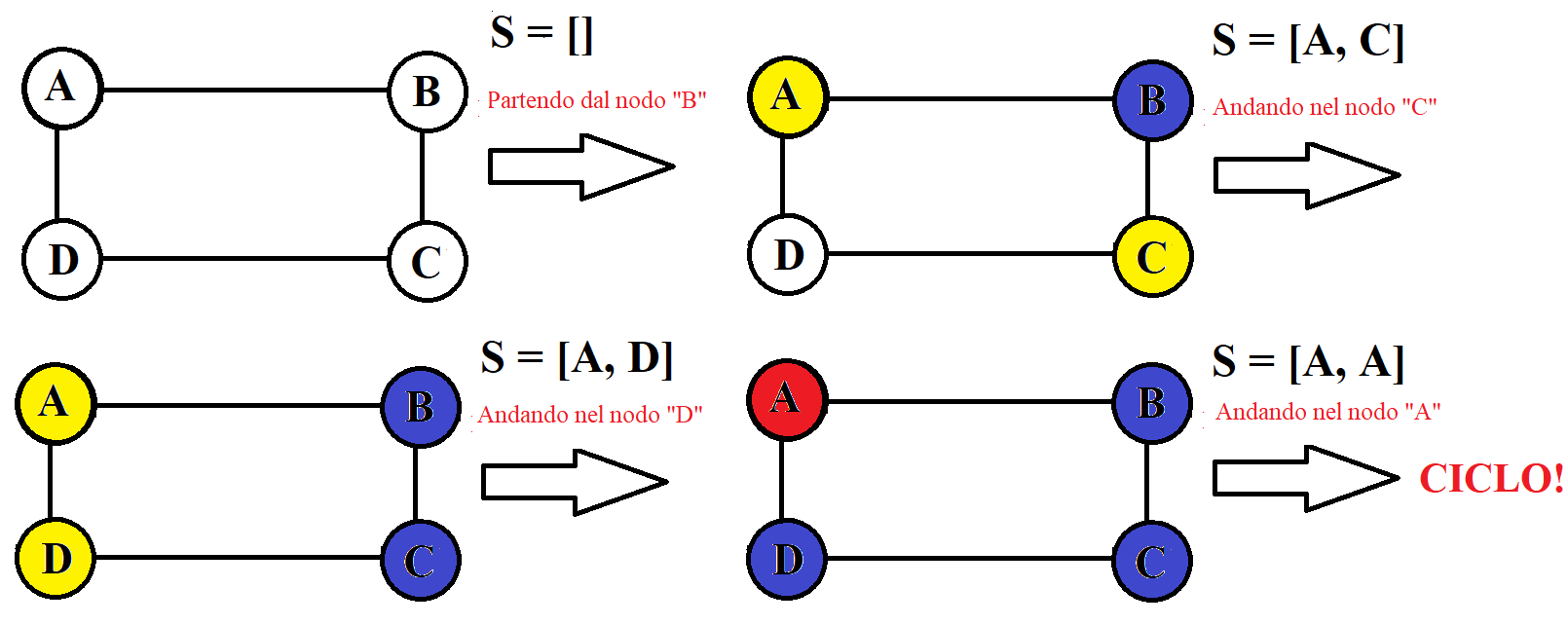


Figura 2. Caso di ciclo in cui i nodi D e B sono connessi tramite un altro nodo C

Implementazione di HCUF

L’algoritmo HCUF, oltre ai parametri opzionali debug e showProfile, prende come input un grafo G e verifica se al suo interno sia presente almeno un ciclo sfruttando la struttura dati Union-Find. Tra le varianti disponibili è stato scelto di implementare la versione QuickFindBalanced, in quanto velocizza al massimo l’operazione di find riducendo al minimo l’impatto prestazionale sull’operazione di union. Inoltre, analizzando lo pseudocodice fornito nella Traccia si può notare come a ogni passo vengano effettuate due operazioni di find e una di union. Dunque, essendo la find l’operazione più frequentemente impiegata, si è rivelata come la più meritevole di ottimizzazione, a scapito della union.

L’algoritmo scorre tutti gli archi del grafo usando un iterabile generato dalla funzione edgeGenerator(). Ad ogni passo, controlla se i valori della coda e della testa dell’arco corrente siano presenti all’interno di un set della Union-Find, e nel caso crea dei set che contengono come unico elemento tali valori (uno per set). Inoltre, richiama edgeGenerator() per generare un iterabile che scorre gli archi finora visitati, in modo da poter controllare se l’arco corrente sia già stato esaminato nel senso opposto. Se non si trova in tale caso, e la testa e la coda dell’arco si trovano nello stesso set, allora l’algoritmo ha trovato un ciclo e termina l’esecuzione, altrimenti tenta di unirne i set e continua con l’esecuzione.

Data la natura dell’implementazione della struttura dati Union-Find è stato necessario definire una nuova classe CustomQFB(QuickFindBalanced) che implementasse il metodo findNode. Tutti i metodi forniti da QuickFindBalanced richiedono in input un nodo UFBNode, dunque è stata ritenuta necessaria l’implementazione di un metodo findNode() che, preso in input un elemento, restituisse il nodo nella Union-Find che lo contenesse. Sfruttando il fatto che gli elementi corrispondono agli ID dei vertici di un grafo e che ogni ID è unico si evita di incorrere in ambiguità, poiché tutti i nodi contengono elementi diversi tra loro.

Il caso peggiore si ottiene nel caso in cui il grafo sia connesso aciclico, dunque disponga di archi. Il tempo di esecuzione in tale caso è .

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 e software 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 metodo Sort di python.

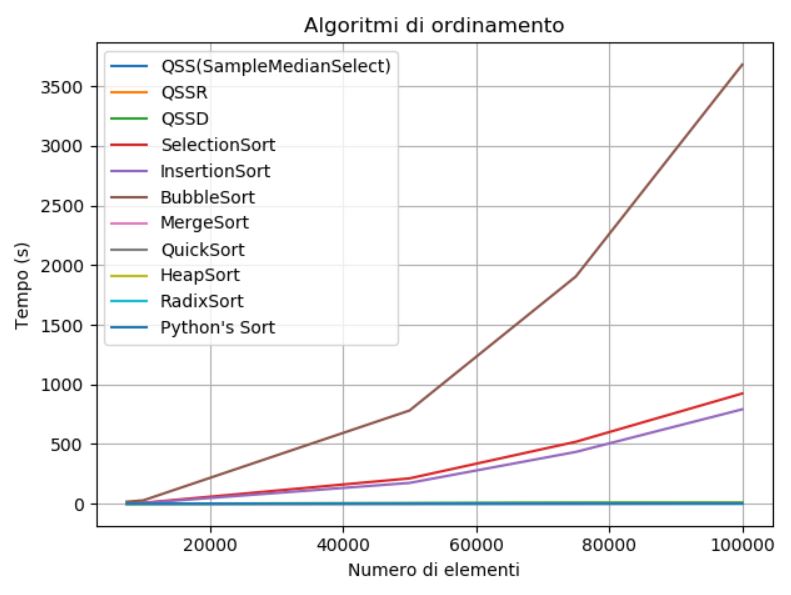


Figura 2. Tempo di esecuzione degli algoritmi al variare della dimensione della lista. Misurazioni riportate nella tabella sottostante.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ELEMENTI | ALGORITMI DI ORDINAMENTO | | | | | | | | | | |
| **No. Elem** | **QSS** | **QSSR** | **QSSD** | **Selection** | **Insertion** | **Bubble** | **Merge** | **Quick** | **Heap** | **Radix** | **Sort** |
| **500** | 0,028 | 0,009 | 0,025 | 0,019 | 0,013 | 0,056 | 0,006 | 0,004 | 0,010 | 0,008 | 0,000 |
| **2500** | 0,112 | 0,049 | 0,138 | 0,444 | 0,338 | 1,483 | 0,027 | 0,018 | 0,057 | 0,300 | 0,001 |
| **5000** | 0,232 | 0,106 | 0,302 | 1,832 | 1,475 | 6,023 | 0,059 | 0,038 | 0,125 | 0,058 | 0,001 |
| **7500** | 0,342 | 0,164 | 0,496 | 3,967 | 3,354 | 16,386 | 0,102 | 0,071 | 0,234 | 0,103 | 0,002 |
| **10000** | 0,454 | 0,220 | 0,680 | 7,350 | 5,619 | 27,983 | 0,139 | 0,098 | 0,318 | 0,136 | 0,002 |
| **50000** | 2,454 | 1,216 | 4,079 | 212,297 | 174,216 | 781,234 | 0,855 | 0,526 | 1,923 | 0,700 | 0,022 |
| **75000** | 4,145 | 2,727 | 8,107 | 519,311 | 434,142 | 1905,262 | 1,298 | 0,786 | 3,064 | 1,077 | 0,034 |
| **100000** | 6,130 | 3,002 | 10,580 | 923,770 | 791,319 | 3681,031 | 2,602 | 1,409 | 5,449 | 1,837 | 0,078 |

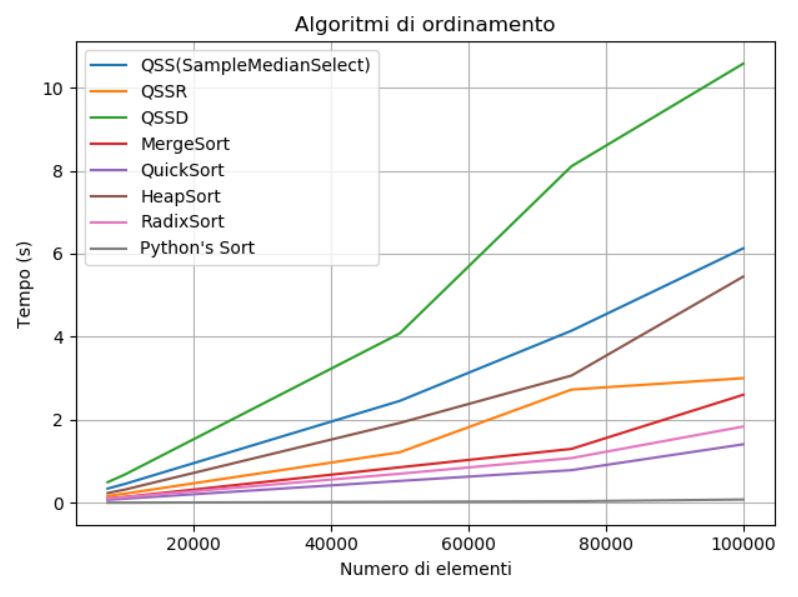


Figura 3. Zoom Figura 2.

Nel caso in cui, la lista in input sia ordinata al contrario, si possono avere delle variazioni nei tempi di esecuzione di algoritmi come il SS e il BS. Infatti il caso peggiore di questi ultimi si presenta proprio in questo caso. Come si può vedere dal grafico, invece, gli altri algoritmi non subiscono variazioni rispetto al tempo calcolato con input casuali.

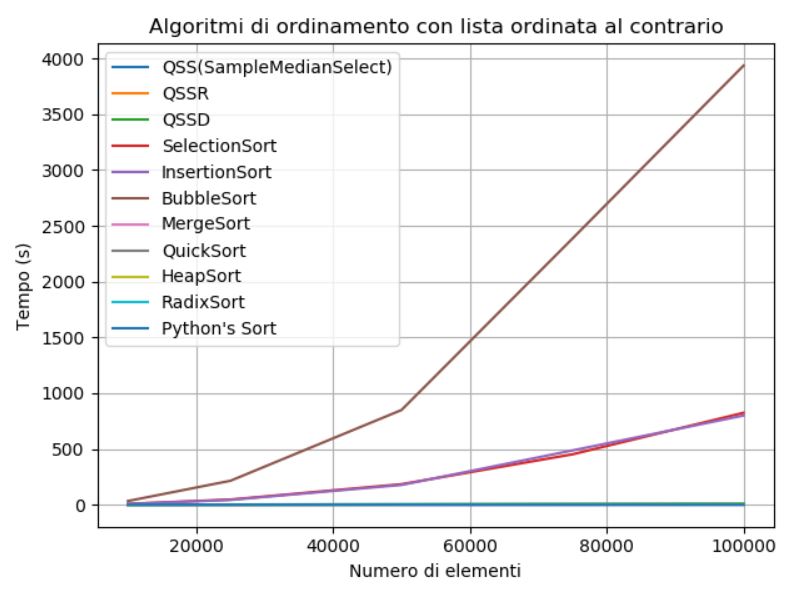


Figura 4. Tempo di esecuzione degli algoritmi al variare della dimensione della lista con input ordinato al contrario. Misurazione riportate nella tabella sottostante.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ELEMENTI | ALGORITMI DI ORDINAMENTO CON LISTA ORDINATA AL CONTRARIO | | | | | | | | | | |
| **No. Elem** | **QSS** | **QSSR** | **QSSD** | **Selection** | **Insertion** | **Bubble** | **Merge** | **Quick** | **Heap** | **Radix** | **Sort** |
| **500** | 0,032 | 0,009 | 0,024 | 0,020 | 0,015 | 0,081 | 0,004 | 0,003 | 0,010 | 0,006 | 0,000 |
| **2500** | 0,123 | 0,059 | 0,135 | 0,546 | 0,486 | 2,061 | 0,023 | 0,018 | 0,056 | 0,030 | 0,000 |
| **5000** | 0,223 | 0,104 | 0,279 | 1,910 | 1,777 | 8,446 | 0,049 | 0,036 | 0,124 | 0,059 | 0,000 |
| **7500** | 0,349 | 0,155 | 0,447 | 4,224 | 3,799 | 18,532 | 0,072 | 0,053 | 0,190 | 0,087 | 0,001 |
| **10000** | 0,429 | 0,212 | 0,583 | 7,468 | 6,621 | 32,992 | 0,095 | 0,072 | 0,254 | 0,114 | 0,001 |
| **50000** | 2,513 | 1,134 | 3,653 | 183,702 | 176,954 | 848,697 | 0,609 | 1,018 | 1,807 | 0,978 | 0,010 |
| **75000** | 4,007 | 1,839 | 6,250 | 451,490 | 486,439 | 2385,772 | 0,928 | 0,626 | 2,573 | 0,936 | 0,010 |
| **100000** | 5,132 | 2,534 | 8,231 | 823,890 | 799,289 | 3935,520 | 1,269 | 0,896 | 3,700 | 1,260 | 0,013 |

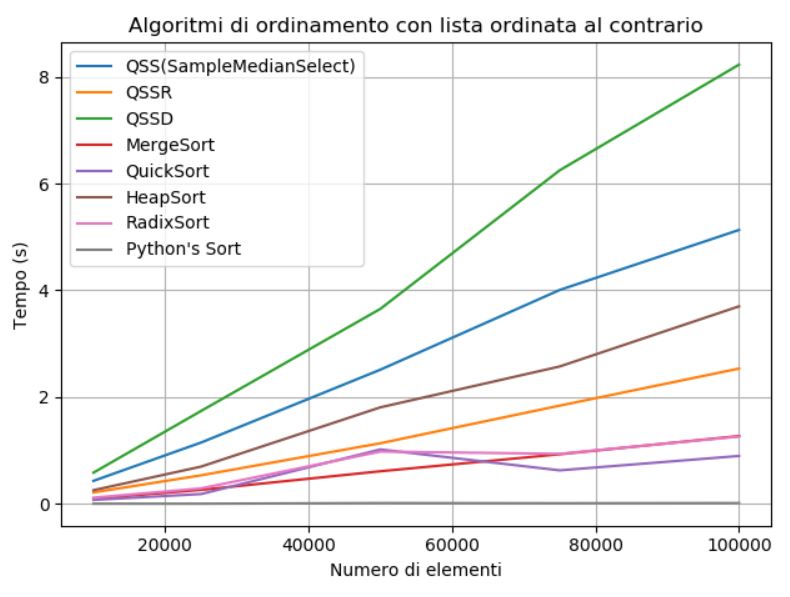


Figura 5, Zoom figura 4.

Nelle figure successive 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. 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.

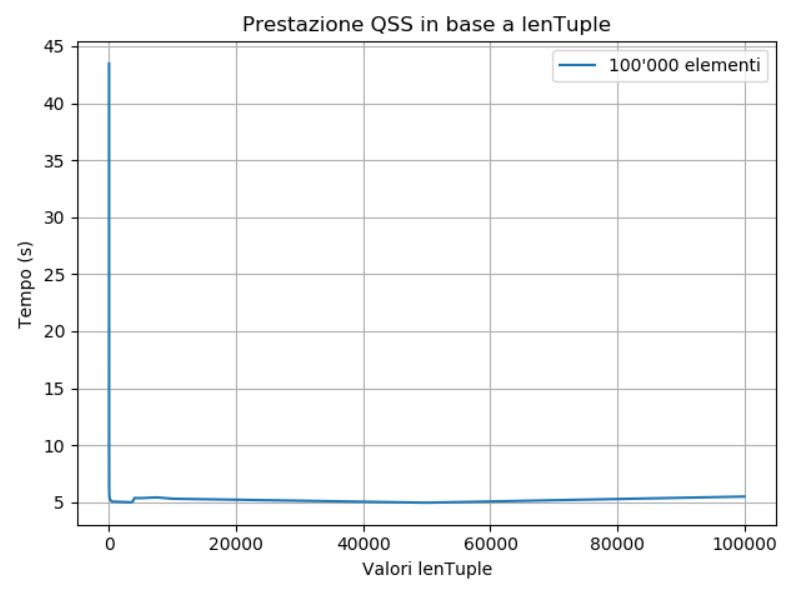


Figura 6. Cambiamento tempi di esecuzione al variare di lenTuple con size costante. (100.000 elementi). Misurazioni riportate nella tabella sottostante.

|  |  |  |  |
| --- | --- | --- | --- |
| **lenTuple** | **Secondi** | **lenTuple** | **Secondi** |
| 1 | 43,472 | 1000 | 5,038 |
| 3 | 19,696 | 2000 | 5,019 |
| 5 | 11,434 | 3500 | 4,988 |
| 7 | 9,348 | 3750 | 5,082 |
| 9 | 8,401 | 4000 | 5,363 |
| 17 | 6,663 | 5000 | 5,352 |
| 25 | 5,906 | 7500 | 5,415 |
| 50 | 5,595 | 10000 | 5,298 |
| 100 | 5,296 | 50000 | 4,951 |
| 200 | 5,205 | 100000 | 5,49 |
| 500 | 5,04 |  |  |

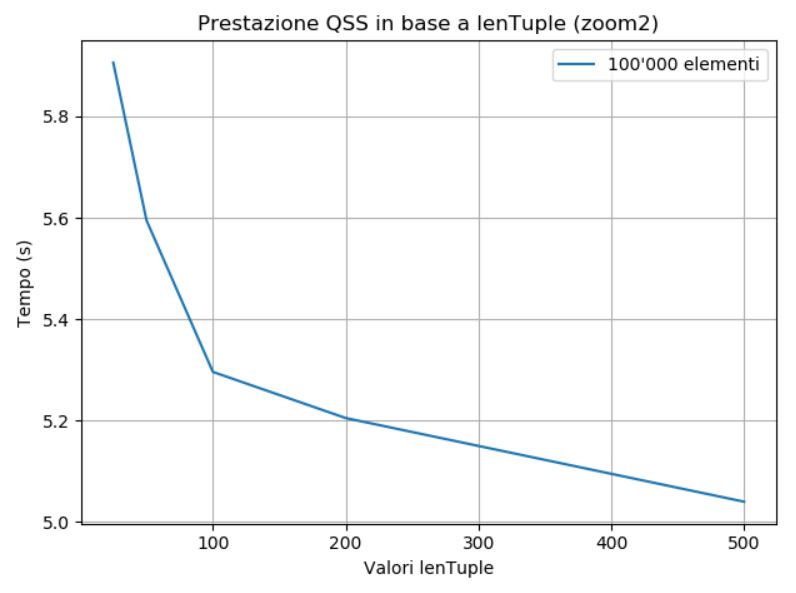
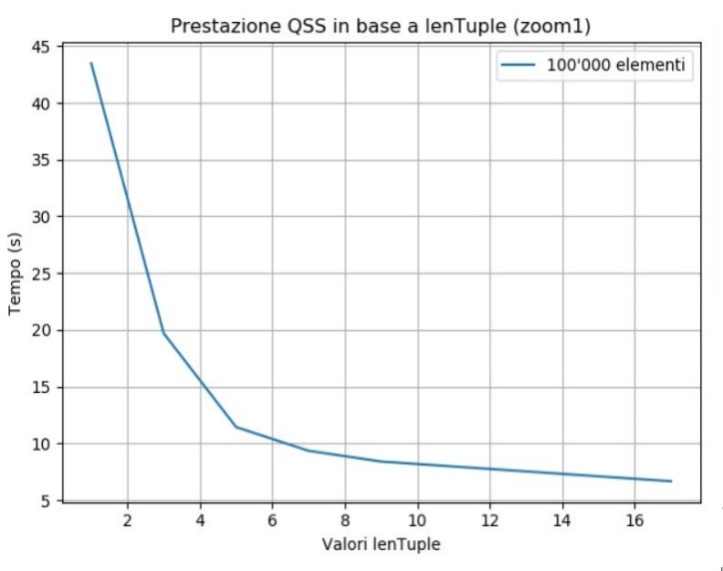


Figura 7. Zoom per valori di lenTuple tra 1 e 17. Figura 8. Zoom per valori di lenTuple tra 25 e 500.

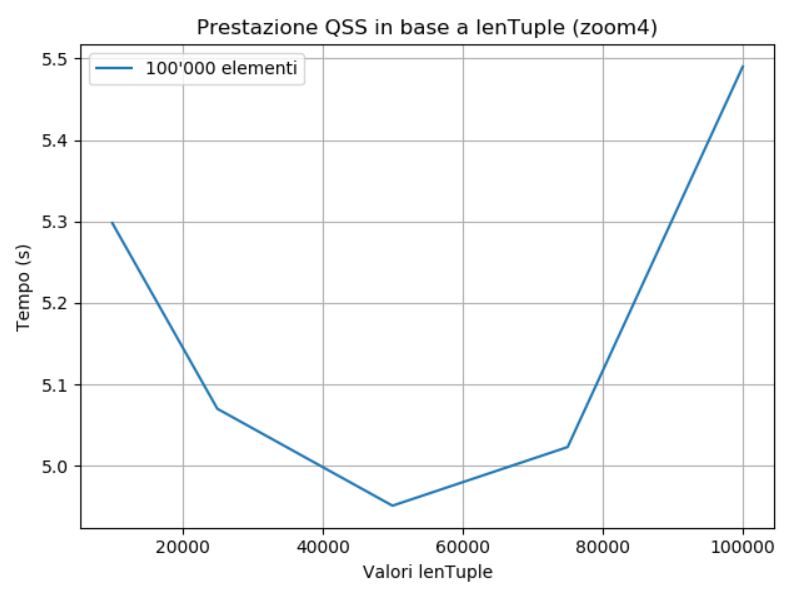
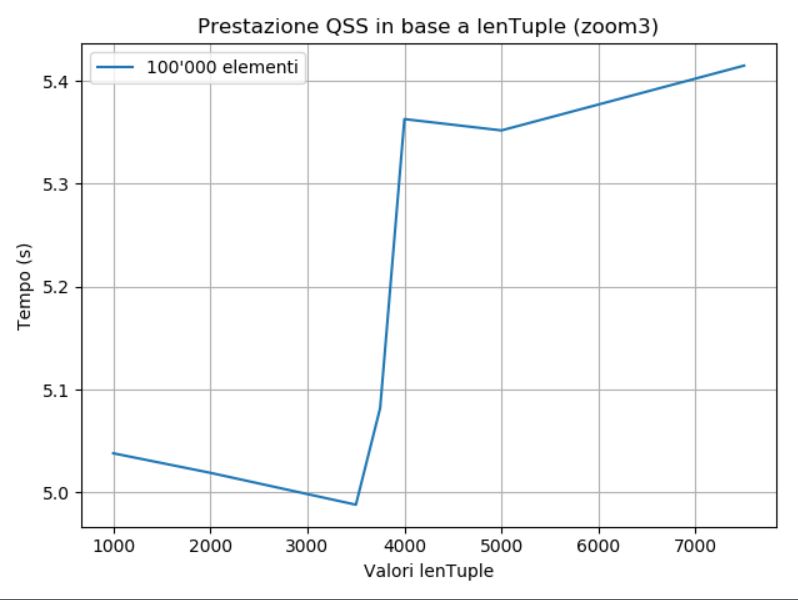


Figura 9. Zoom per valori di lenTuple tra 1000 e 7500. Figura 10. Zoom per valori di lenTuple tra 10.000 e 100.000.

Un ulteriore fattore che influenza il tempo di esecuzione di QSS 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.

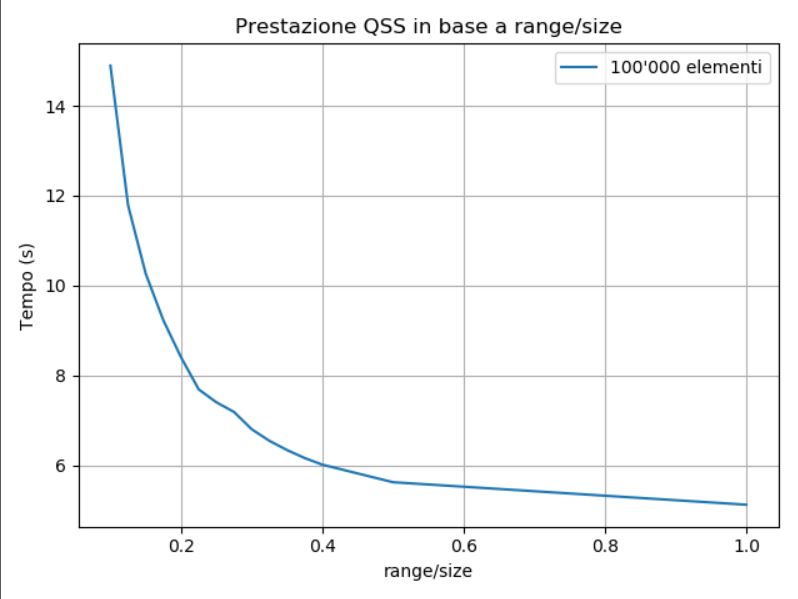


Figura 11.Tempo di esecuzione al variare del rapporto . Misurazioni riportate

nelle tabelle sottostanti.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Range/Size** | **Secondi** |  | **Range/Size** | **Secondi** |  |
| 1 | 5,126 |  | 0,25 | 7,404 |  |
| 0,5 | 5,626 |  | 0,225 | 7,69 |  |
| 0,4 | 6,016 |  | 0,2 | 8,404 |  |
| 0,375 | 6,164 |  | 0,175 | 9,225 |  |
| 0,35 | 6,341 |  | 0,15 | 10,262 |  |
| 0,325 | 6,547 |  | 0,125 | 11,785 |  |
| 0,3 | 6,804 |  | 0,1 | 14,893 |  |
| 0,275 | 7,188 |  |  |  |  |

Un ultimo test è stato effettuato su una lista in input ordinata per il 90% dei suoi elementi. Si può notare dal grafico in Figura 12 come in questo caso l’IS ed il BS presentino una diminuzione nel tempo di esecuzione. Ciò è dato dal fatto che la lista data in input rappresenta per essi il caso migliore che si possa verificare.

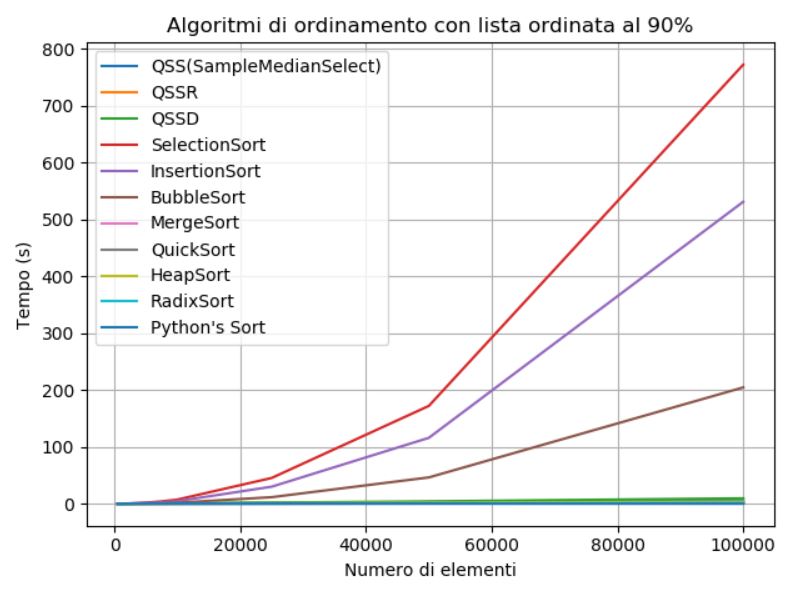


Figura 12. Tempo di esecuzione con lista ordinata al 90%. Misurazioni riportate nella tabella sottostante.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ELEMENTI | ALGORITMI DI ORDINAMENTO CON LISTA ORDINATA AL 90% | | | | | | | | | | |
| **No. Elem** | **QSS** | **QSSR** | **QSSD** | **Selection** | **Insertion** | **Bubble** | **Merge** | **Quick** | **Heap** | **Radix** | **Sort** |
| **500** | 0,033 | 0,010 | 0,023 | 0,018 | 0,010 | 0,004 | 0,004 | 0,003 | 0,011 | 0,007 | 0,000 |
| **2500** | 0,125 | 0,049 | 0,127 | 0,447 | 0,272 | 0,106 | 0,025 | 0,006 | 0,023 | 0,012 | 0,000 |
| **5000** | 0,244 | 0,099 | 0,273 | 1,820 | 1,116 | 0,412 | 0,050 | 0,018 | 0,061 | 0,029 | 0,000 |
| **7500** | 0,413 | 0,156 | 0,452 | 4,000 | 2,443 | 0,933 | 0,074 | 0,035 | 0,133 | 0,057 | 0,000 |
| **10000** | 0,444 | 0,207 | 0,613 | 7,222 | 4,472 | 1,729 | 0,105 | 0,055 | 0,205 | 0,088 | 0,001 |
| **50000** | 2,632 | 1,170 | 4,068 | 171,862 | 115,770 | 46,189 | 0,582 | 0,453 | 1,652 | 0,580 | 0,003 |
| **75000** | 3,780 | 1,729 | 5,530 | 400,461 | 252,352 | 108,075 | 0,906 | 0,595 | 2,520 | 0,870 | 0,004 |
| **100000** | 5,426 | 2,542 | 9,424 | 771,678 | 530,642 | 204,351 | 1,568 | 0,869 | 3,663 | 1,246 | 0,006 |

Approfondimenti

In questa sezione sono approfonditi diversi aspetti del materiale trattato durante la realizzazione del progetto.

Cenni Storici

L’algoritmo QuickSort fu sviluppato nel 1959 da Tony Hoare durante la sua permanenza nell’Unione Sovietica come studente in visita alla *Moscow State University*. All’epoca, Hoare stava lavorando a un progetto riguardante un software di traduzione per il *National Physical Laboratory*. Come parte del progetto, egli cercò di ideare un algoritmo che ordinasse alfabeticamente le parole all’interno di frasi scritte in russo, in modo da velocizzarne la ricerca in un dizionario Russo – Inglese. Resosi conto che la sua prima scelta, InsertionSort, presentava tempi di esecuzione eccessivamente lunghi, implementò un algoritmo del tutto nuovo, ovvero il QS.

Egli pubblicò il codice di QS su un articolo di *Communications of the Association of Computing Machinery*, la più prestigiosa rivista di ingegneria informatica del tempo. In seguito, l’algoritmo ottenne una larghissima adozione, apparendo per esempio nell’OS **Unix** come la predefinita libreria di ordinamento.

Il principio di funzionamento di QS trova applicazione anche in problemi di **selezione**. Una sua variante, il QuickSelect (Tony Hoare), opera in modo simile per estrarre un certo elemento  da liste disordinate.

*Fonte: https://en.wikipedia.org/wiki/Quicksort*