Relazione esercizi progetto Laboratorio di Algoritmi e Strutture Dati

Abourida Zakaria – 950120

Eusebio Bergò Stefano - 952388

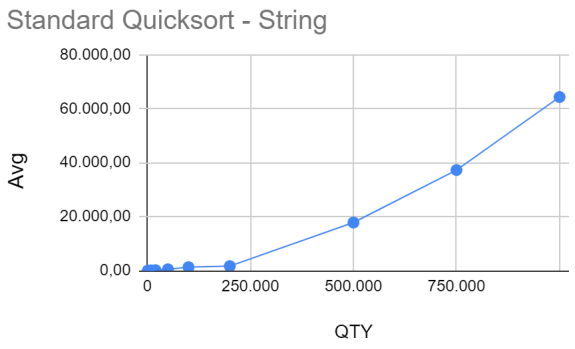
Primo esercizio

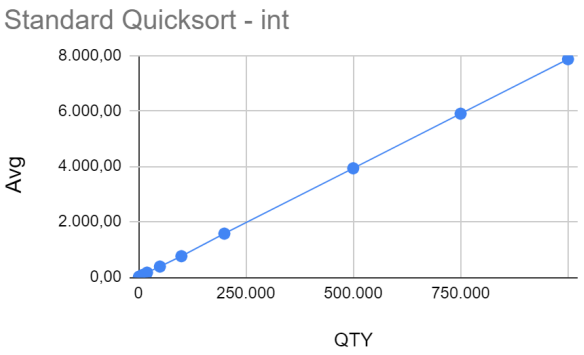
I test sono stati effettuati su input di 1k, 2k, 5k, 10k, 20k, 50k, 100k, 200k, 500k, 750k, 1000k elementi ed infine sul file da 20 milioni di elementi. (Il tempo segnato è in millisecondi.)

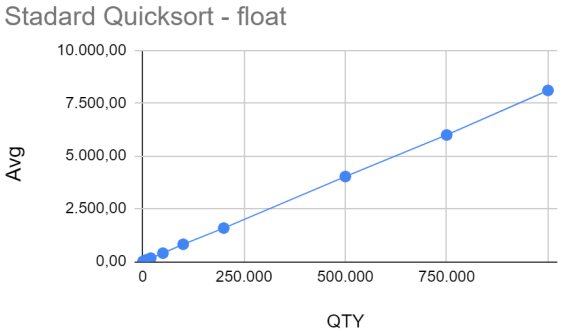
- Tempo medio di caricamento dei dati nella struttura per un input da 20 milioni 1min 27sec.

Quicksort

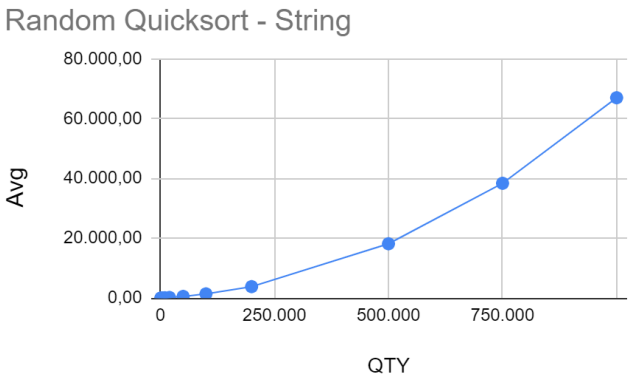
test di quicksort con il pivot sull'ultimo elemento (Standard) dell'array (X = quantità, Y = tempi medi):

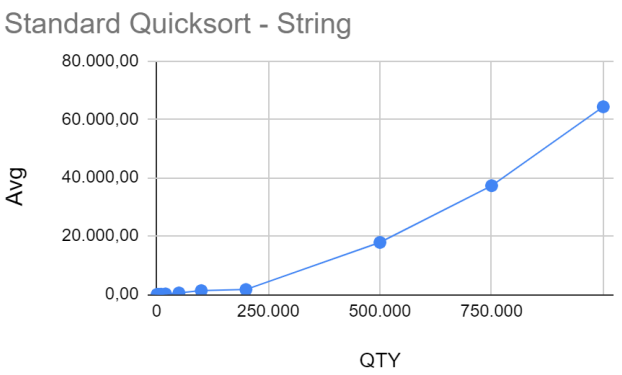






Questo Algoritmo ha complessità temporale nel caso medio O(n * log n) e O(n^2) nel caso peggiore. Abbiamo notato che durante l'esecuzione dell'algoritmo sui campi String impiega un maggior tempo per la dimensione del dato da analizzare, il confronto appartiene ad O(min(a.lenght, b.lenght)). Per i campi numerici, integer e float, il confronto ha complessità O(1) in quanto calcola la differenza tra i due dati.
La crescita asintotica dei tempi medi di esecuzione non sembra essere legata alla tipologia dell'input scelto.

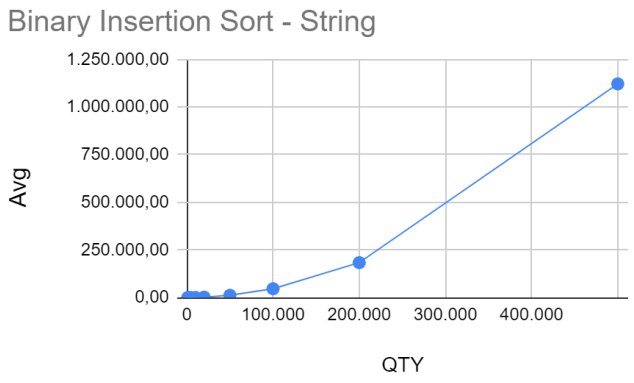


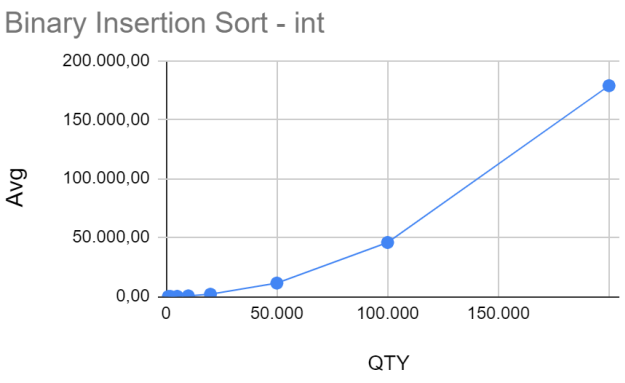


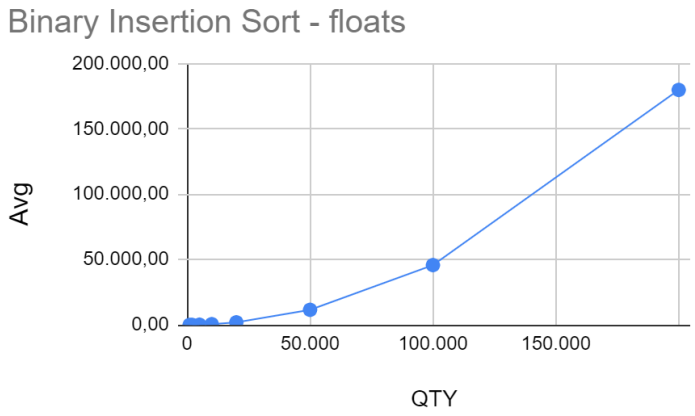
Per quanto concerne la scelta dell'elemento di pivot abbiamo notato che la scelta non influisce molto sui tempi in quanto equivale ad aggiungere uno scambio tra l'elemento scelto ed il pivot di base e poi applicare l'algoritmo di base. Il rischio di scegliere il primo o l'ultimo elemento come elemento di pivot è quello per cui ci si possa poi ritrovare nel caso peggiore (in caso l'array di partenza sia già ordinato). Una soluzione sarebbe quella di utilizzare come pivot un elemento non estremo, per esempio un elemento casuale (ma potrebbe capitare l'estremo) o meglio ancora l'elemento centrale in modo tale da evitare i casi peggiori .
Si potrebbe voler scegliere come elemento di pivot il massimo/minimo dell'array ma questo implica una precedente ricerca di tale elemento (in O(n)) e quindi un possibile aumento di complessità.

Binary Insertion Sort

dati dei test sui vari tipi di input(X = quantità, Y = tempi medi):





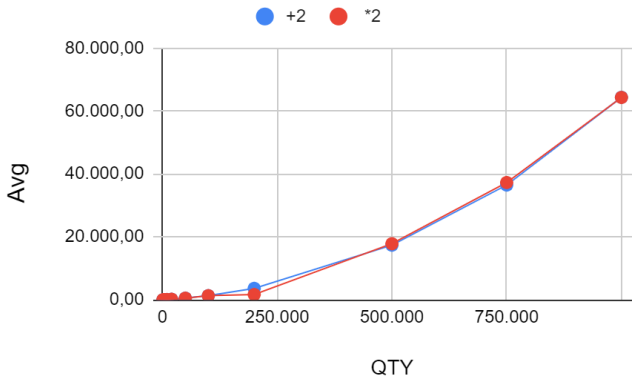


In questi grafici i test si fermano a quantità inferiori rispetto ai precedenti in quanto i tempi di esecuzione superano i 10 min e quindi, come indicatoci, sono stati ignorati.

Questo algoritmo parte di base dall'InsertionSort. La differenza è che utilizziamo la ricerca dicotomica. Abbiamo visto che con questa variante si riesce a ridurre la complessità temporale della ricerca della posizione in cui inserire ogni elemento da O(n) a O(log n). Come per il Quicksort, i confronti sulle stringhe sono più onerosi in quanto dipendono dalla lunghezza dei due elementi confrontati. La crescita asintotica dei tempi di ogni tipo di campo è molto simile.

Ridimensionamento dell'array

Abbiamo svolto due test sul metodo di ridimensionamento della struttura (array) utilizzato negli algoritmi precedenti.
Il primo test consiste nel calcolo della nuova dimensione dell'array moltiplicando la dimensione attuale per una costante (nell'esempio 2), mentre il secondo consiste nel calcolo della nuova dimensione incrementando la dimensione attuale di una costante (nell'esempio 2).

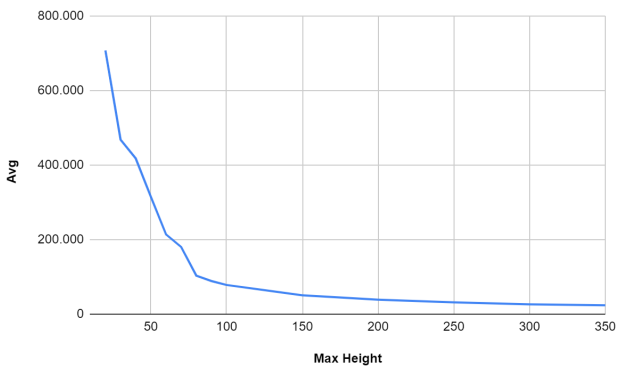


Nonostante ci aspettassimo che il primo metodo risultasse più efficiente, dal grafico si può notare che i due metodi si equivalgono. Questo probabilmente è causato dall' "esiguo" numero di elementi presi in analisi, probabilmente non abbastanza sufficienti da far notare una differenza significativa.

Secondo Esercizio

	time																			
max height	min.	millisec.	min.	millisec.	min.	millisec.	min.	millisec.	min.	millisec.	min.	millisec.	min.	millisec.	min.	millisec.	min.	millisec.	min.	millisec.	Average
10																					
20	12	22259	12	10057	13	14258	13	31197	13	8305	13	11471	13	59520	13	7867	12	34955	12	24563	707.677
30	8	35593	8	29430	8	30280	8	58434	8	46017	8	31152	8	33563	8	36446	8	51819	7	58070	468.255
40	7	10042	6	23273	7	28606	8	10762	8	16740	7	49596	7	20527	7	54703	8	991	8	2950	418.017
50	5	59360	5	51045	5	45823	5	26741	5	39575	4	27471	4	56992	6	4791	7	26768	6	4876	314.858
60	3	42141	3	41805	3	47775	3	39571	3	52157	4	39452	4	41247	3	35266	3	43649	3	51147	214.019
70	3	28684	3	24466	3	23999	3	22733	3	17518	3	2793	3	16429	3	19796	3	33488	2	53017	180.266
80	1	50953	1	57368	1	51684	1	52252	1	52281	1	54263	1	57760	1	53174	1	55721	1	51833	103.390
90	1	36821	1	39160	1	37919	1	36938	1	36700	1	40716	1	37285	1	39680	1	34838	1	36986	88.822
100	1	25476	1	24654	1	28330	1	21890	1	26805	1	27202	1	25849	1	26096	1	26932	1	27950	78.289
150	0	59148	0	54870	0	54589	0	56787	0	55747	0	56864	0	54006	0	53757	0	56845	0	53986	50.600
200	0	39341	0	42973	0	46612	0	44566	0	46241	0	44662	0	42799	0	41160	0	40087	0	39282	38.884
250	0	35698	0	36745	0	35745	0	33886	0	36888	0	36686	0	35340	0	34101	0	31859	0	33089	31.822
300	0	31683	0	27612	0	28792	0	30361	0	28263	0	28574	0	27668	0	29275	0	29248	0	29464	26.449
350	0	25599	0	25472	0	26767	0	26555	0	27605	0	25784	0	26722	0	26318	0	25679	0	26419	23.902

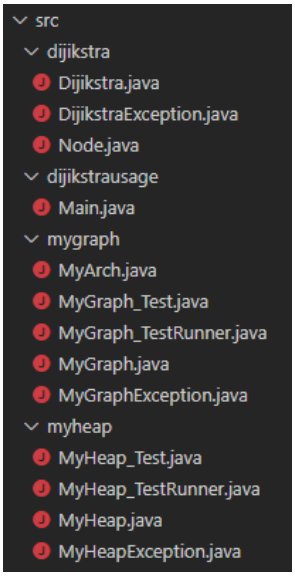
Questi sono i risultati ottenuti tramite il test di vari valori come altezza massima permessa alla SkipList. Come indicatoci, i tempi di gran lunga superiori ai 10 minuti non sono stati considerati; per questo la prima riga non contiene valori.



Dalla tabella e dal grafico si può dedurre che con l'aumento dei livelli le tempistiche si abbassano notevolmente.

Per i tempi medi si può intuire un limite asintotico superiore di 1/MaxHeight.

Terzo e Quarto Esercizio



L'esercizio è stato diviso in tre pacchetti base (myheap, mygraph e dijikstra) che contengono le relative classi funzionali e classi di UnitTest.

E' anche presente un pacchetto "dijikstrausage" che contiene la classe eseguibile principale, ovvero quella che si relaziona con dijikstra.Dijikstra caricando i dati del grafo, richiedendo i nomi delle città all'utente, richiedendo il calcolo delle distanze poi stampando i valori da prendere in considerazione.

La classe myheap.MyHeap implementa un heap min/max e le funzioni di base richieste per una struttura heap. Per permettere a queste operazioni una complessità consona all'heap è stata utilizzata una HashTable (che ha complessità O(1) sulle operazioni) ed un

Vector. L'HashTable fornisce un'associazione tra un elemento e l'indice all'interno del Vector in cui è memorizzato.

Lo stesso metodo è stato utilizzato nella classe mygraph.MyGraph, in cui viene implementata la struttura di un grafo orientato (un grafo non orientato viene implementato tramite l'inserimento sia degli archi di "andata" che quelli di "ritorno"), per memorizzare i nodi del grafo. Gli archi vengono memorizzati in una HashTable che fornisce l'associazione tra un dato nodo di partenza ed un elenco dei nodi adiacenti; l'elenco dei nodi adiacenti è anch'esso rappresentato con una HashTable che fornisce l'associazione tra il nodo destinazione ed il peso associato all'arco nodo_partenza → nodo_destinazione. Per indicare gli adiacenti di un nodo, si è scelto di utilizzare una HashTable di HashTable invece di una HashTable di List in quanto è nettamente più veloce nelle operazioni relative agli archi (inserimento, rimozione, modifica e ottenimento).

Per il metodo che restituisce l'elenco degli archi attualmente presenti è stata implementata una mappa che contiene l'arco (nodo_partenza, nodo_destinazione) ed il peso associato; questo in quanto è stato richiesto di svolgere questo metodo rispettando un tempo massimo O(n).

Sono riportati di seguito dei dati ottenuti tramite l'algoritmo di dijikstra per le rispettive coppie di città (i percorsi non sono orientati in quanto in questo esercizio i cammini si possono percorrere sia in un senso che nell'altro):

torino – milano = 495.479 Km

torino – ivrea = 51.176 Km

milano – ivrea = 508.416 Km

bardonecchia – brindisi = 1088.241 Km

roma – venezia = 416.184 Km