PROGETTO ALGORITMI A.A. 2021/22

COMPONENTI:
Emanuele Carlucci, 950023
Federico di Geronimo 945865

ESERCIZIO 1:

Sono stati implementati due algoritmi di ordinamento:
-Binary-InsertionSort
-QuickSort e la sua variante Randomized-QuickSort

-Randomized-QuickSort

La variante random dell'algoritmo QuickSort riduce la complessità dal suo caso peggiore $O(n^2)$ al suo caso medio O(nlogn), modificando la scelta del pivot da arbitraria (primo elemento) a random.
Scegliere deterministicamente il pivot (caso QuickSort) può portare nel caso di input particolari (array ordinati) a creare delle partizioni sbilanciate, in cui una delle due è di lunghezza n - 1 e l'altra di lunghezza 0.
Scegliendo un pivot random invece, a parità di input non necessariamente verranno a formarsi partizioni completamente sbilanciate, e questo approssima la complessità del Randomized-QuickSort al caso medio, cioè O(nlogn).

-Ipotesi:

Randomized-QuickSort quindi, dovrebbe terminare in tempi inferiori rispetto a QuickSort nel caso in cui l'array è già ordinato.
Nel caso in cui l'array non causa partizioni sbilanciate la complessità del QuickSort è uguale a quella del Randomized-QuickSort, anche se ci aspettiamo tempi di terminazione differenti in quanto, il secondo deve svolgere le operazioni di scelta del pivot e quindi avrà una costante moltiplicativa maggiore.

-Osservazioni:

Testando i due algoritmi sull'input fornito su GIT(20 mln di record), abbiamo rilevato una media di 8.7s (su 5 tentativi) nel caso di QuickSort e una media di 8.9s (su 5 tentativi) nel caso di Randomized-QuickSort.
Questo è il risultato previsto, in quanto Randomized-QuickSort deve svolgere delle operazioni in più rispetto al QuickSort.
Per validare le nostre ipotesi riguardo al motivo di questa differenza abbiamo testato i due algoritmi sullo stesso input, questa volta ordinato, causando quindi partizioni sbilanciate.
Abbiamo rilevato che QuickSort non termina mentre Randomized-QuickSort termina in tempi medi.
Queste osservazioni provano le nostre ipotesi in quanto rileviamo che QuickSort performa molto peggio di Randomized-QuickSort.

ESERCIZIO 2

Per implementare la funzione di insert di un elemento nella skiplist, si utilizza la funzione randomlevel() che ritorna il numero di livelli per quel determinato elemento.
Questa funzione non restituisce un semplice numero casuale, ma restituisce un livello n tra 0 e MAX_HEIGHT con probabilità $(1 - ½)^n$ (distribuzione geometrica).

Scegliere il livello di un elemento con una distribuzione geometrica invece che con la tipica rand() serve per meglio distribuire i nodi con un alto numero di livelli.

ESERCIZIO 4

Per rappresentare il grafo si è deciso di utilizzare una lista di adiacenza.
Si è preferito utilizzare una lista di adiacenza al posto di una matrice di adiacenza per i seguenti motivi:

1) Una matrice nel caso di grafi sparsi risulta essere molto meno ottimale perchè a prescindere dal numero di archi occupa in memoria sempre O(nodi^2);
2) Per trovare gli adiacenti per mezzo di una matrice bisogna scorrere tutti i possibili vicini, O(n). Con una lista di adiacenza trovare i vicini è triviale in quanto ogni nodo li punta direttamente.

Per implementare la lista di adiacenza è stata utilizzata un' ArrayList contenente delle LinkedList, che rappresentano effettivamente la lista degli adiacenti del nodo contenuto nella testa dell'array.
Inoltre per poter accedere ad ogni nodo con tempo costante sono stati salvati gli indici dell'ArrayList in una HashTable.
I grafi diretti e indiretti sono identificati tramite un numero intero 0 (diretto), 1 (indiretto).
Nel caso di grafi indiretti viene aggiunto il vertice di arrivo e il suo peso nella lista degli adiacenti del nodo di andata e viceversa.

Per realizzare l'algoritmo di Dijkstra si è deciso di utilizzare un grafo, una coda a priorità minima, realizzata con un MinHeap, ed infine un ArrayList di vertici.

Compilazione degli esercizi:
-Esercizio 1: La compilazione avviene tramite il comando make. Dopo essersi posizionati nella cartella Esercizio1 basta eseguire: ./bin/Sort_main "file_di_input.csv" "file_di_output.csv" "campo da ordinare" "algoritmo di ordinamento".
Per i test basta eseguire: make testsSortinglib / make testsDynamicArraylib

-Esercizio 2: La compilazione avviene tramite il comando make. Dopo essersi posizionati nella cartella Esercizio2 basta eseguire: ./bin/skip_list_main "dizionario.txt" "file_da_correggere.txt"
Per i test basta eseguire: make tests

-Esercizio 3: La compilazione avviene tramite il comando ant all. Dopo essersi posizionati nella cartella Esercizio3 basta eseguire: java -jar ./build/HeapMinimo.jar “file_di_input.txt” .
Per i test basta eseguire: java -jar ./build/HeapMinimo_Test.jar

-Esercizio 4: La compilazione avviene tramite il comando ant all. Dopo essersi posizionati nella cartella Esercizio4 basta eseguire: java -jar ./build/Dijkstra.jar input_file.csv .
Per il test del grafo basta eseguire: java -jar ./build/GrafoTests.jar
Per il test dell'heapminimo basta eseguire: java -jar ./build/HeapMinimoTests.jar