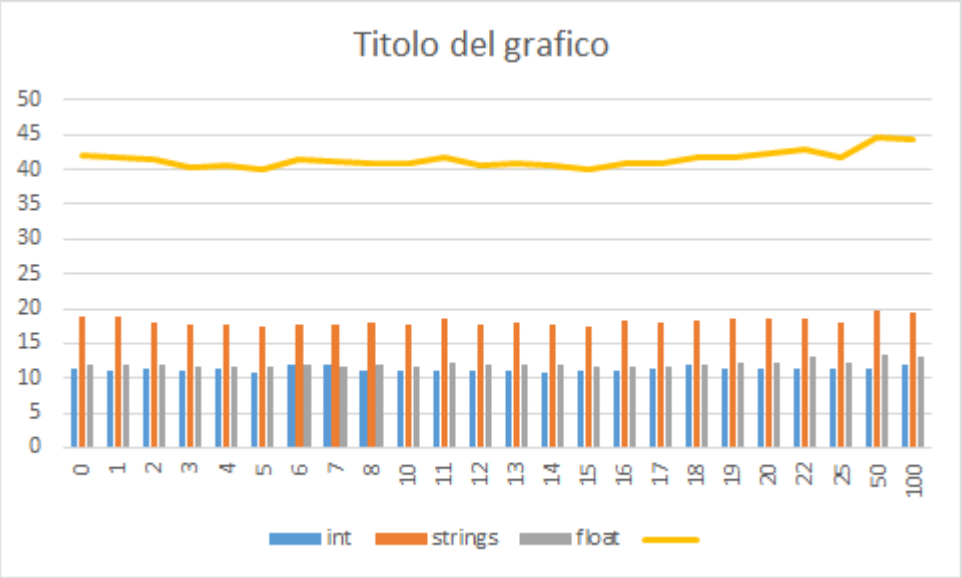
Studente: Haures Daniel
Matricola: 919707
Appello: 01/07/2021

Relazione Laboratorio di Algoritmi e Strutture Dati

Esercizio 1

I risultati ottenuti dall'algoritmo di sorting sono i seguenti:

k	int	strings	float	total
0	11,486	18,743	11,947	42,176
1	11,084	18,749	11,898	41,731
2	11,346	18,06	11,953	41,359
3	10,94	17,737	11,582	40,259
4	11,268	17,788	11,643	40,699
5	10,853	17,501	11,596	39,950
6	11,853	17,682	11,853	41,388
7	11,885	17,602	11,567	41,054
8	11,168	17,943	11,850	40,961
10	11,18	17,831	11,742	40,753
11	11,008	18,691	12,152	41,851
12	10,972	17,646	11,884	40,502
13	11,018	17,896	11,865	40,779
14	10,882	17,800	11,846	40,528
15	11,086	17,295	11,730	40,111
16	11,114	18,308	11,574	40,996
17	11,296	17,886	11,741	40,923
18	11,801	18,145	11,869	41,815
19	11,252	18,435	12,165	41,852
20	11,410	18,686	12,112	42,208
22	11,489	18,441	12,956	42,886
25	11,490	17,915	12,280	41,685
50	11,412	19,783	13,450	44,645
100	11,866	19,369	13,186	44,421
1000	17,840	24,480	18,197	60,517
5000	62,399	69,356	62,638	194,393
10000	111,365	120,24	114,688	346,293
20000	213,574	221,925	214,293	649,792
100000	826,331	835,918	827,000	2489,249



Il binary insertion sort ha complessità temporale quadratica mentre il merge sort ha complessità O(k log k), quindi teoricamente all'aumentare di k anche i tempi di calcolo dovrebbero aumentare.
Tuttavia ciò non avviene, i tempi di calcolo dei valori k da 2 a 17 dimostrano di essere addirittura leggermente inferiori dei valori k=0 e k=1, rappresentanti l'algoritmo senza binary insertion sort.
Tale situazione è possibile perchè nonostante l'algoritmo di *binary insertion sort* nel caso peggiore dimostra complessità $O(k^2)$, nel caso migliore è O(klogk).
Per valori k piccoli l'influenza dei casi peggiori sui casi minori è ridotta e nel complesso l'algoritmo ha complessità simile a O(k log k).
Se invece assumiamo valori k grandi, i casi peggiori rendono ininfluenti i casi minori, rendendo preferibile la sola esecuzione dell'algoritmo di merge sort.
A dimostrazione di ciò, da k=18 in poi i tempi di calcolo iniziano a crescere e superano quelli di k=0 e k=1, arrivando addirittura a rendere inutilizzabile l'algoritmo.
Si può notare anche che sebbene i tempi di calcolo da k=2 a k=17 siano generalmente inferiori, essi non seguono un ordine preciso, questo è dovuto all'ordine iniziale dell'array, che a seconda di k può generare una distribuzione incostante di casi peggiori e casi migliori.
Risulta perciò necessario scegliere k compreso tra 2 e un valore sufficientemente piccolo da non rendere k=0 e k=1 preferibili.

Esercizio 2

Risultati:

```
quando-->[quando]
avevo-->[avevo]
cinqve-->[cinque,cive]
anni-->[anni]
mia-->[mia]
made-->[made]
mi-->[mi]
perpeteva-->[erpete,permetteva,perpendeva,perpetra,perpetrava,perpetrera,perpetua
,perpetuava,perpetue,perpetuera,repeteva,ripeteva]
sempre-->[sempre]
che-->[che]
la-->[la]
felicita-->[felicita]
e-->[e]
la-->[la]
chiave-->[chiave]
della-->[della]
vita-->[vita]
quando-->[quando]
andai-->[andai]
a-->[a]
squola-->[suola]
mi-->[mi]
domandrono-->[domandarono]
come-->[come]
vuolessi-->[volessi]
essere-->[essere]
da-->[da]
grande-->[grande]
io-->[io]
scrissi-->[scrissi]
selice-->[selice]
mi-->[mi]
dissero-->[dissero]
che-->[che]
non-->[non]
avevo-->[avevo]
capito-->[capito]
il-->[il]
corpito-->[carpito,clorito,coito,colpito,compito,copiato,copio,copto,corio,corpi,
corpino,corpo,corto,covrito,crepito,scorpio]
e-->[e]
io-->[io]
dissi-->[dissi]
loro-->[loro]
che-->[che]
non-->[non]
avevano-->[avevano]
capito-->[capito]
la-->[la]
wita-->[aita,cita,dita,gita,iota,irta,ista,iuta,lita,pita,sita,ta,vita,witz,zita]

Free
time:11.039000
```

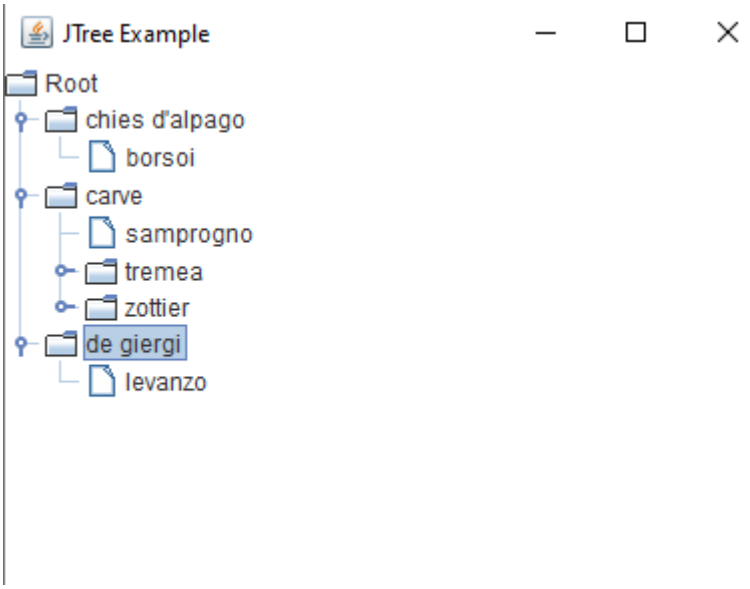
Nel corso della sua ricorsione, l’algoritmo di edit distance può sviluppare chiamate che hanno gli hanno gli stessi parametri di input, rendendo necessario calcolare più volte chiamate di funzione identiche.
Per esempio, se prendessimo le parole “massa” e “mese”, l'algoritmo ripeterebbe la chiamata “assa” e “ese” sia dopo aver scelto di non fare alcuna operazione, che dopo aver scelto di eseguire un'operazione di inserimento e una di cancellazione.
Risulterebbe quindi efficiente per i tempi di calcolo l’implementazione della programmazione dinamica.
L’algoritmo ricorsivo sviluppato con la programmazione dinamica ha riportato un tempo di calcolo nettamente superiore rispetto all’algoritmo non dinamico.
La correzione dell’intera frase con l’algoritmo dinamico richiede all’incirca 11 secondi su windows, mentre l’algoritmo normale ha richiesto 1 ora e 30 minuti(5424.72 secondi).
La causa principale di un tempo di esecuzione così grande nell’algoritmo normale è da attribuirsi alla sua complessità temporale esponenziale, che viene invece evitata nell’algoritmo dinamico.
A dimostrazione di ciò, durante l’esecuzione dell’algoritmo senza programmazione dinamica, parole molto lunghe come “*perpeteva*” e “*domandrono*” hanno richiesto la quasi totalità del tempo necessario all’esecuzione.

Esercizio 3

Per l'implementazione dell'Union Find Set sono state utilizzate due HashMap.
La prima HashMap prende come chiave un elemento e vi associa un oggetto di tipo Node, il quale rappresenterà le relazioni dell'elemento con i nodi degli altri elementi.
La seconda HashMap invece prende come chiave l'oggetto nodo e vi associa elemento che esso rappresenta.
La creazione di un insieme prevede perciò la creazione di un nodo per l'elemento passato in input e l'inserimento nelle due hashmap della relazione nodo-elemento.
Le operazioni di Union e FindSet opereranno quindi internamente sui nodi degli elementi, sfruttando le associazioni presenti nelle due HashMap.

Esercizio 4

Risultati:




```
Totale nodi:18640
Totale archi prima dell'algoritmo di Kruskal: 48055
Totale archi dopo l'algoritmo di Kruskal: 18637
Somma in km: 89939,9126
Tempo di caricamento grafo: 180ms
Tempo di esecuzione Kruskal: 172ms
```

La struttura dati grafo è stata realizzata tramite una hashmap esterna e una hashmap interna, in modo da poter favorire la complessità temporale.

La hashmap esterna associa un nodo di tipo generico a una seconda hashmap interna, che rappresenta la lista delle adiacenze del nodo.
La hashmap interna contiene come chiave i nodi che sono adiacenti al nodo-chiave della prima hashmap e come valore l'etichetta dell'arco espresso dai due nodi-chiave, anche essa di tipo generico.
La creazione di un nodo perciò consiste nell'inserimento del nodo nella hashmap esterna e alla sua associazione con una lista di adiacenza.
L'inserimento di un arco avviene invece tramite l'inserimento del nodo adiacente nella hashmap interna associata al nodo di partenza.
Gli archi non sono perciò rappresentati da un unico oggetto nella struttura interna e per facilitare l'inserimento e la restituzione degli archi è stato necessario creare una classe di supporto Edge, che contiene il nodo di partenza, il nodo di arrivo e l'etichetta associata all'arco.
L'algoritmo di kruskal estrae tutti gli archi creati sotto forma di un arraylist di oggetti di tipo Edge, li ordina e crea un altro grafo contenente solo gli archi che formano la MST.
La somma totale dei km descritti nelle etichette degli archi è 89939,9126 e l'algoritmo di kruskal è stato eseguito in 172 ms.
La classe contenente l'algoritmo di kruskal dispone anche di un arraylist che rappresentano le possibili radici della foresta generata.