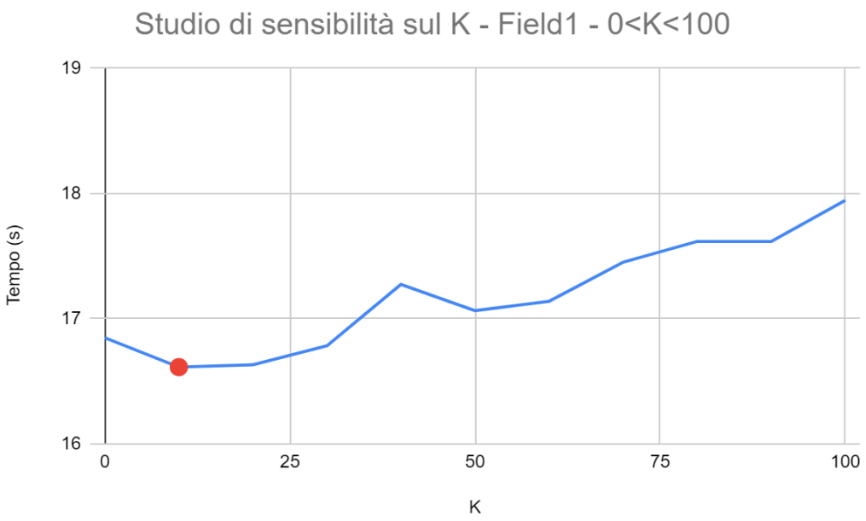
RELAZIONE ASD

Esercizio 1: Merge-BinaryInsertion Sort

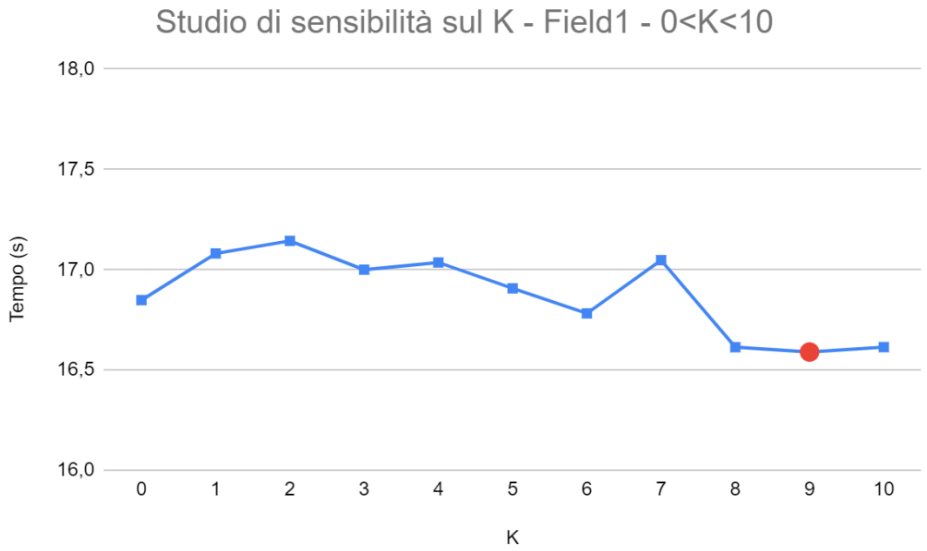
In questo studio, sono stati misurati i tempi di risposta di un algoritmo di ordinamento utilizzando tre diversi campi come chiave di ordinamento: *String, Interi e Floating point*. Il valore di k è stato variato, e per ciascun valore di k, sono stati registrati i tempi di ordinamento. Di seguito, riporteremo i risultati ottenuti per ogni caso e commenteremo i risultati.

Field1 – String:

K	*TEMPO(s)*
0	16,848236
10	16,613922
20	16,632708
30	16,783964
40	17,274168
50	17,065359
60	17,139368
70	17,451265
80	17,617777
90	17,616987
100	17,945536
MINIMO	**16,613922**



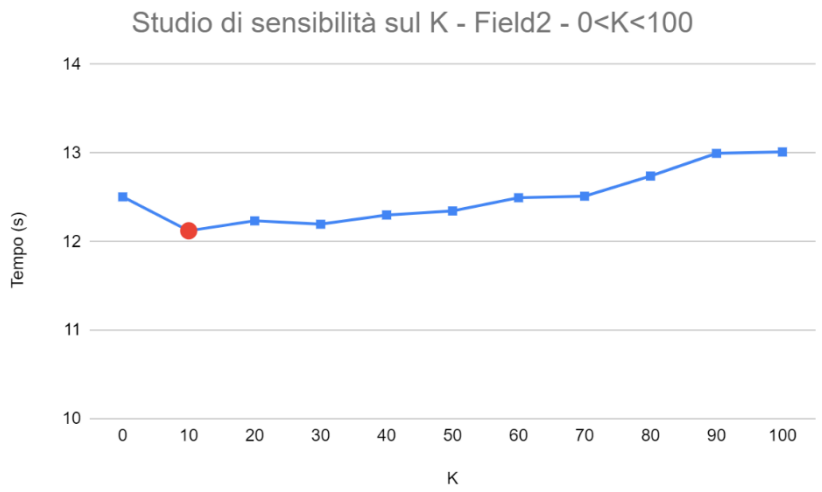
0	16,848236
1	17,080931
2	17,143663
3	17,000067
4	17,036055
5	16,906998
6	16,782078
7	17,047604
8	16,613462
9	16,589361
10	16,613922
MINIMO	**16,589361**



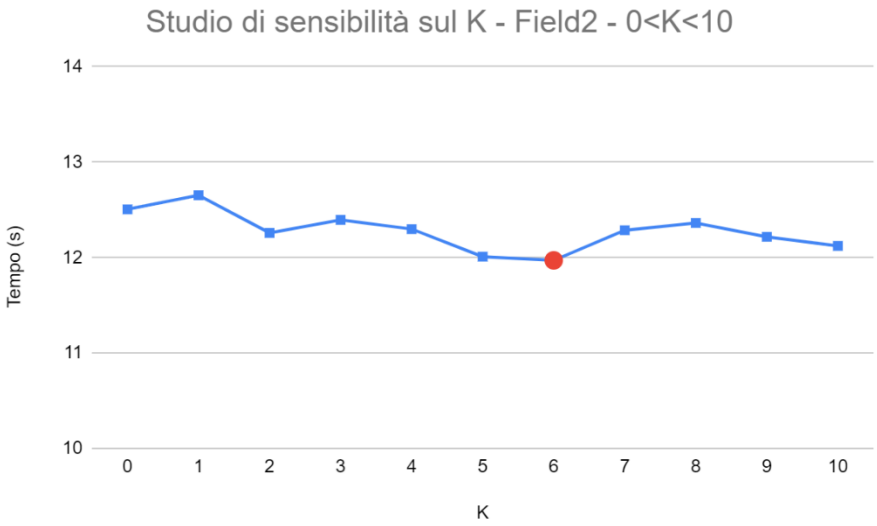
Field2 – Intero:

K	*TEMPO(s)*
0	12,505338
10	12,122112
20	12,234456

30	12,19628
40	12,300736
50	12,346119
60	12,495008
70	12,512692
80	12,739997
90	12,996311
100	13,012129
MINIMO	**12,122112**

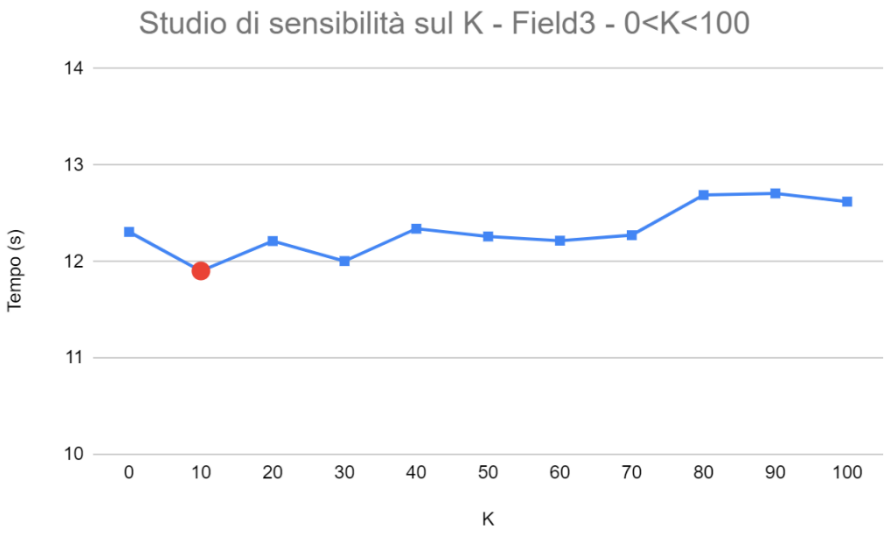


0	12,505338
1	12,653154
2	12,258606
3	12,395161
4	12,298093
5	12,009708
6	11,970734
7	12,286225
8	12,362876
9	12,217317
10	12,122112
MINIMO	**11,970734**

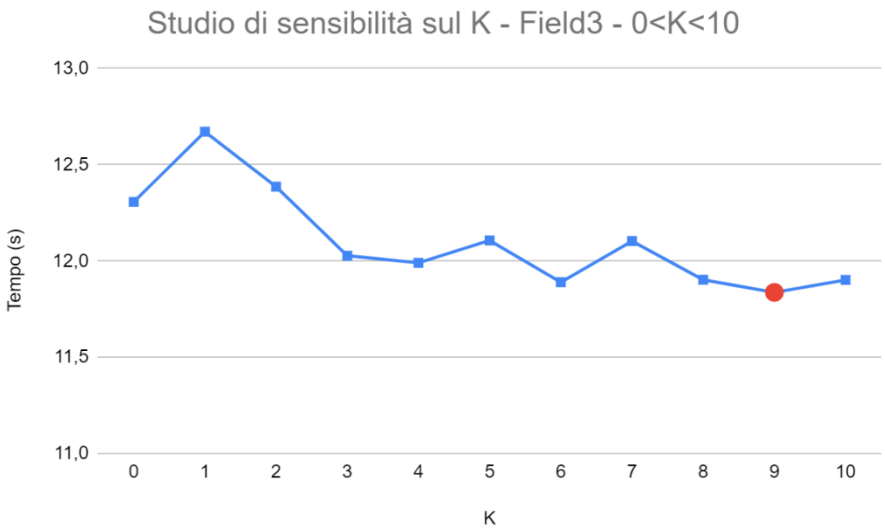


Field3 – Floating point:

K	***TEMPO(s)***
0	12,30694
10	11,901654
20	12,211844
30	12,004421
40	12,33988
50	12,259532
60	12,215911
70	12,273045
80	12,689018
90	12,706247
100	12,620904
MINIMO	**11,901654**



0	12,30694
1	12,671741
2	12,386166
3	12,02741
4	11,990475
5	12,106829
6	11,889951
7	12,103652
8	11,90257
9	11,836906
10	11,901654
MINIMO	**11,836906**



I grafici mostrano come il tempo di esecuzione varia in funzione del tempo. Evidenziano anche l'esistenza di differenti valori di k ottimali per i diversi campi utilizzati come criterio di ordinamento. Al fine di massimizzare l'efficienza del nostro algoritmo Merge-BinaryInsertion Sort e ottenere tempi di ordinamento minimi, è necessario mantenere valori di k bassi. Infatti con k elevati, l'algoritmo BinaryInsertion Sort impiega molto tempo per eseguire l'ordinamento.

D'altra parte però, se si riduce troppo il valore di k, si applica l'algoritmo Merge Sort anche su un array di dimensioni ridotte, anche se l'Insertion Sort è quello più efficiente per tali casi. Pertanto, è importante trovare un valore di k ottimale che bilanci l'efficienza dell'Insertion Sort per array piccoli e la velocità complessiva dell'ordinamento.

Nel nostro caso specifico, i valori di k ottimali sono i seguenti:
- *Field1 – String*: il valore di k ottimale è 9
- *File2 – Interi*: il valore di k ottimale è 6.
- *Fild3 – Floating point*: il valore di k ottimale è 9.

È importante notare che all'aumentare del valore di k in tutti e quattro i grafici, il tempo di ordinamento aumenta notevolmente, causando di fatto un impatto negativo sull'efficienza dell'algoritmo. Pertanto, la scelta di un valore di k adeguato è fondamentale per ottenere prestazioni ottimali nell'ordinamento dei dati.

Utilizzo:

Per compilare eseguire il programma posizionarsi nella cartella *Ex_1* ed eseguire i seguenti comandi:

- Compilare: `make all`
- Eseguire i test: `make runtest`
- Eseguire il programma: `make runmain ARGS="Parametro1 Parametro2 …"`

  In particolare il programma richiederà un file di input, un file di output, un valore k per l'algoritmo di ordinamento e un valore per il field secondo cui ordinare

Esercizio 2: Skip-List

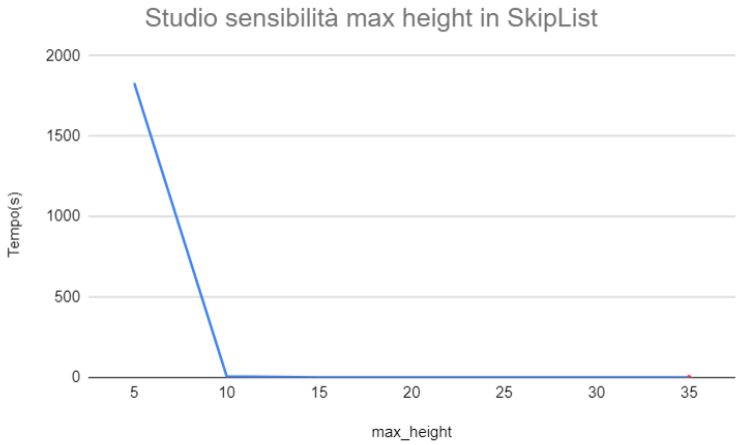
Nel seguente studio sono stati misurati i tempi di caricamento di una skip-list, ovvero una struttura dati che si può scorrere più velocemente delle classiche liste concatenate in quanto ogni nodo possiede non un semplice puntatore all'elemento successivo ma un array di puntatori a diversi elementi successivi.

Lo studio verte sul ruolo della variabile *max_height* che determina l'altezza massima che ogni nodo può raggiungere. Come si può facilmente dedurre ciò andrà ad influire sulle prestazioni di caricamento e lettura della skip-list: avendo nodi che possono guardare a più elementi successivi si può raggiungere più facilmente l'elemento ricercato.

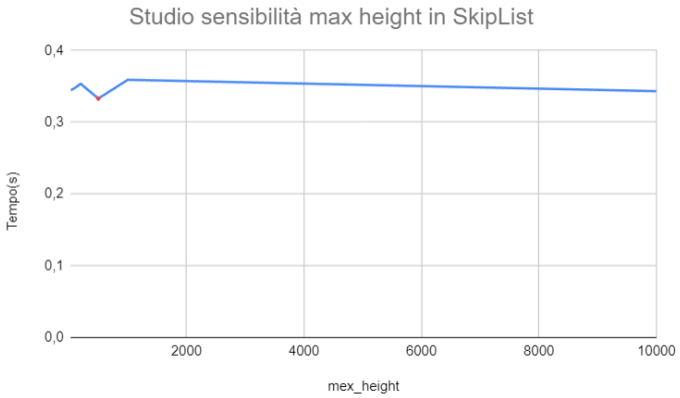
L'altezza di ogni nodo è calcolata in modo randomico; infatti la funzione *random_level()* estrae un numero compreso tra 0 e 1, se il numero è minore di 0.5 l'altezza del nodo continuerà a crescere. Logicamente valori di *max_height* tanto alti è difficile che l'altezza continui a crescere poiché nella funzione *random_level()* si agisce come il lancio di una moneta: testa se il numero è compreso tra 0 e 0.5, croce altrimenti; continuare ad aumentare il livello significherebbe fare tante volte croce consecutivamente e la probabilità suggerisce che è difficile.

Il fatto di usare una funzione *random_level()* permette di avere una skip-list scalabile, infatti modificando i parametri della funzione si può adattare la struttura dati in base ai dati forniti e far si che rimanga sempre bilanciata.

5	1830,176147
10	7,103716
15	0,452014
20	0,390603
25	0,364787
30	0,344567
35	0,341583
MINIMO	**0,341583**



30	0,344567
100	0,347705
200	0,353565
500	0,332962
1000	0,359104
10000	0,343271
MINIMO	**0,332962**



Come si può notare dai grafi con un valore di *max_height* molto elevato il tempo di caricamento dei dati tende a rimanere costante, a confronto di valori molto piccoli che portano tempi anche maggiori di 30 minuti.

Utilizzo:

Per compilare eseguire il programma posizionarsi nella cartella *Ex_2* ed eseguire i seguenti comandi:

- Compilare: `make all`
- Eseguire i test: `make runtest`
- Eseguire il programma: `make runmain ARGS="Parametro1 Parametro2 …"`

In particolare il programma richiederà un file di input con i dati del dizionario, un file di input con il testo da correggere, un valore max_height per l'altezza massima dei singoli nodi

Relazione redatta dagli studenti:

-Candela Davide mat. 976529

-Rastello Umberto mat. 996575

-Manildo Filippo mat. 1061474