UNIVERSITÀ
DI TORINO

Dipartimento di Informatica

Progetto di Algoritmi e Strutture Dati

977654	Bruno Luca	luca.bruno767@edu.unito.it
976965	Van Cleef Jacopo	jacopo.vancleeff@edu.unito.it

0. Introduzione

Il progetto verte sullo studio, sviluppo ed implementazione di i algoritmi e strutture dati accompagnati da uno studio di complessità temporale. Di seguito vengono riportati gli algoritmi e le strutture dati che verrano analizzate ed i linguaggi utilizzati:

1. Merge-BinaryInsertion Sort, C
2. Skiplist, C
3. Grafo con coda di priorità, Java

Verrano analizzate le singole implementazioni e verranno discusse le scelte implementative adottate. Verrà poi analizzato come l'efficenza delle implementazioni vari al variare di determinati parametri/condizioni.

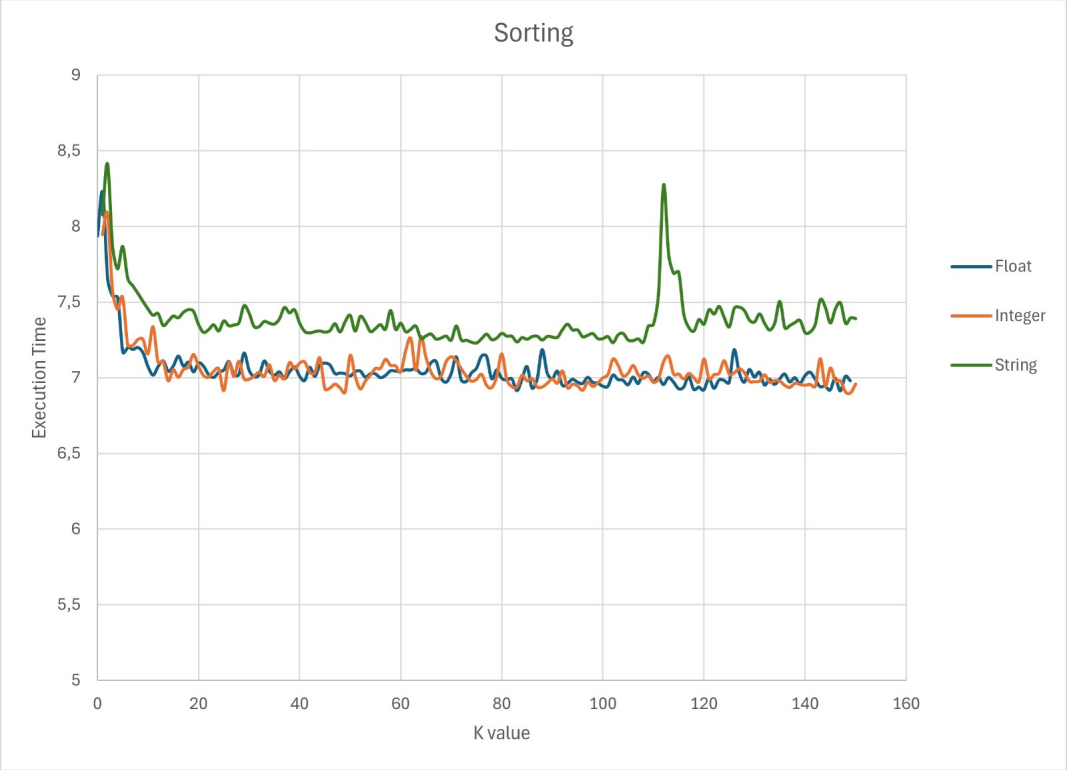
1. Merge-BinaryInsertion Sort

Si vuole implementare una libreria che offre un algoritmo di ordinamento *Merge-BinaryInsertion Sort* su dati generici, secondo il seguente prototipo di funzione:

```
void merge_binary_insertion_sort(void *base, size_t nitems, size_t size, size_t k,
                                 int (*compar)(const void*, const void*));
```

L'algoritmo combina *BinaryInsertion Sort* e *Merge Sort*, sfruttando il fatto che il primo può essere più veloce del secondo quando la lista da ordinare ha una grandezza limitata. Pertanto l'algoritmo inizierà l'ordinamento secondo il tipico funzionamento del MergeSort. Quando però la lunghezza delle sotto liste da ordinare scende sotto il parametro *K* l'algoritmo procede ad ordinare la sotto lista con BinaryInsertion Sort, dove la posizione, all'interno della sezione ordinata del vettore, in cui inserire l'elemento corrente è determinata tramite ricerca binaria.

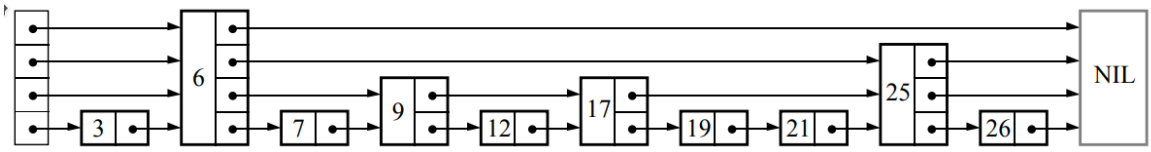
Di seguito viene riportato il tempo di esecuzione dell'algoritmo al variare del parametro k per tutti e tre diversi tipi di variabili:



Come si evince dal grafico con valori di *K* molto piccoli l'algoritmo impiega più tempo di esecuzione. Aumentandone il valore ottimizziamo l'esecuzione, fino a raggiungere un approssimazione asintotica oltre la quale l'aumento di *K* non influisce sulle prestazioni dell'algoritmo. Notiamo inoltre come la scelta del tipo di dato inficia sul tempo di esecuzione. Questo è dovuto dal fatto che il confronto tra dati dipende direttamente dal tipo di dato che stiamo confrontando: interi e float sono meno onerosi rispetto a stringhe.

2. Skiplist

Si vuole implementare una libreria che offre la struttura dati probabilistica Skiplist.



Tale struttura dati permette operazioni di ricerca con complessità temporale O(log n), così come inserimenti e cancellazioni in tempo O(log n), rendendola adatta per l'indicizzazione di dati. La SkipList accelera diverse operazioni creando "vie espresse", consentendo di "skippare" parte della lista. Ogni nodo al suo interno non ha solo un puntatore al prossimo elemento, ma un array di puntatori che permette di saltare a vari punti successivi determinati in modo probabilistico.

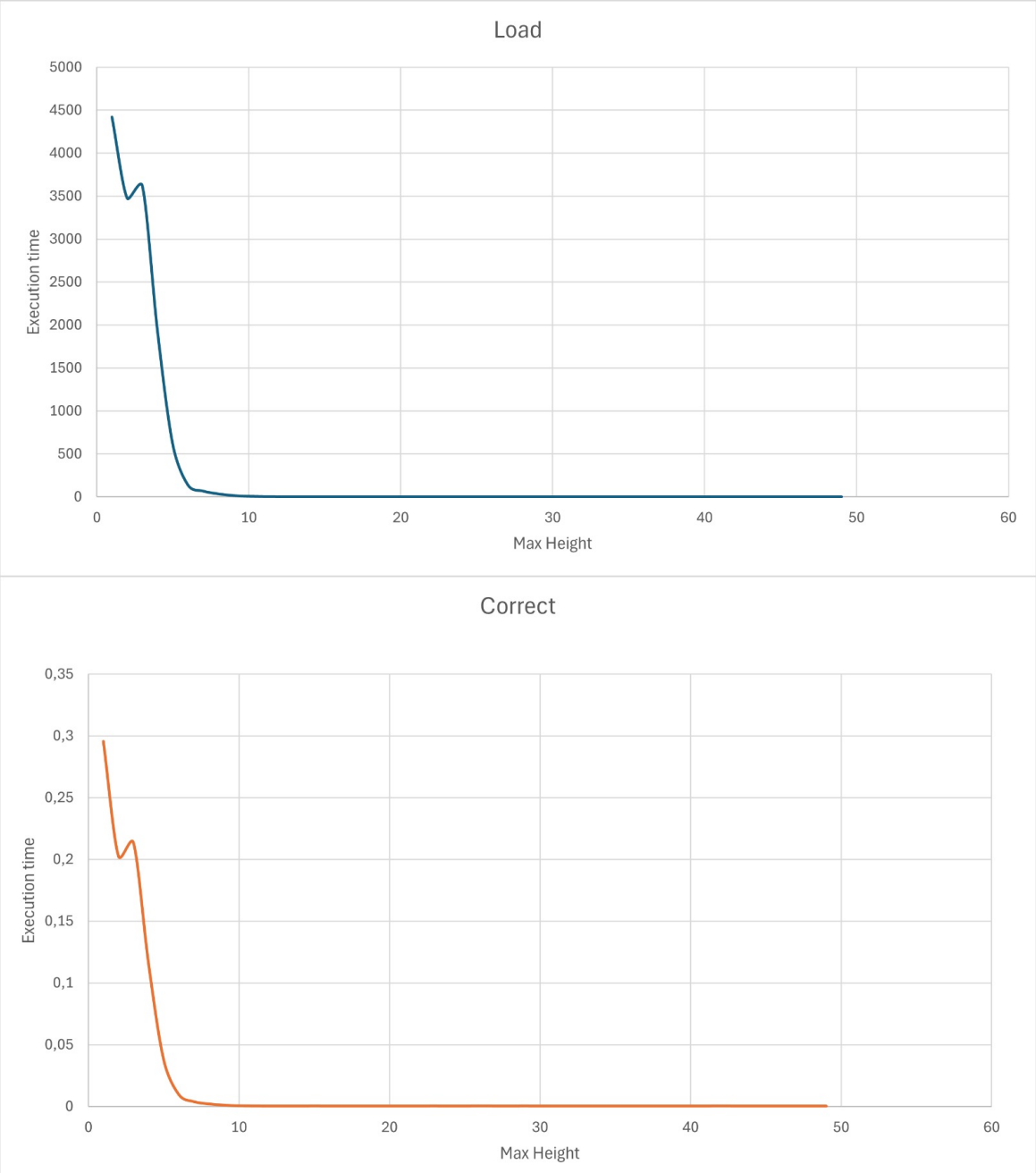
La grandezza della lista dei puntatori viene determinata in maniera casuale dal seguente algoritmo:

```
int randomLevel(int max_height):
    int lvl = 1

    while(random() < 0.5 and lvl < max_height):
        lvl++;

    return lvl
```

Questo algoritmo è particolarmente vantaggioso perché evita di avere all'interno della struttura dati un numero elevato di nodi ad alto livello. La scelta dell'altezza di un nodo è paragonabile ad una prova bernoulliana. Più si sale di livello meno è probabile

che la funzione continui ad aumentarne il valore. In questo modo le vie espresse che si vengono a creare permettono di saltare grandi parti della struttura dati, rendendola più efficiente di una normale lista.

L'efficenza della struttura dati dipende anche in modo diretto dall'altezza massima che viene impostata al momento di inizializzazione. Un'altezza massima bassa causerebbe la presenza di troppi nodi ad alto livello, rallentando di conseguenza tutte le operazioni che si effettuano sulla struttura dati. Questo risulta evidente dal grafico che raffigura il variare del tempo di esecuzione in funzione dell'altezza massima della skiplist:



3. Grafo con PriorityQueue

Stiamo sviluppando l'implementazione di una struttura dati grafo, ottimizzata per offrire prestazioni ottimali in contesti in cui i dati sono sparsi. Questa implementazione è progettata per garantire una massima generalità sia per quanto riguarda i tipi di nodi, per soddisfare una vasta gamma di casi d'uso, sia per quanto riguarda le etichette degli archi. L'obiettivo centrale è fornire una soluzione altamente adattabile.

Parallelamente, ci stiamo concentrando sulla realizzazione di una libreria dedicata alla struttura dati coda con priorità (PriorityQueue) con un focus primario sulla gestione fluida di tipi generici e sulla flessibilità nell'adattarsi dinamicamente a un numero variabile e indefinito di elementi.

La struttura PriorityQueue è rappresentata tramite un ArrayList, sebbene l'accesso non sia in tempo costante (O(1)) a causa della sua natura intrinseca. Per ovviare a questa limitazione, è stata affiancata da una HashMap, dove ogni elemento della coda funge da chiave e l'indice della sua posizione nell'ArrayList rappresenta il valore associato.

Per quanto riguarda la struttura dati grafo, è stata implementata attraverso l'utilizzo di una HashMap, in cui i nodi del grafo sono utilizzati come chiavi. A ciascuna chiave è associato un HashSet che memorizza gli archi che partono da quel nodo specifico. Il HashSet funge da contenitore di elementi unici, senza un ordine specifico. Si comporta essenzialmente come una scatola in cui è possibile inserire vari oggetti, ma garantendo l'unicità di ciascuno. Inoltre, è ottimizzato per eseguire ricerche rapide al fine di verificare la presenza di un elemento specifico all'interno della collezione.