**Laboratorio 1 – Sorting efficiente del file loader**

Questo codice in C++ legge un array di numeri interi da un file, ordina l'array utilizzando l'algoritmo di ordinamento Shell Sort, e misura il numero di operazioni di lettura effettuate durante il processo. Successivamente, stampa le statistiche sulle operazioni di lettura nei casi migliore, medio e peggiore.

Le variabili ct\_swap, ct\_cmp e ct\_read sono contatori per il numero di swap, confronti e letture effettuate durante l'ordinamento.

Le variabili max\_dim, ntests, ndiv, details, graph e n sono variabili di configurazione.

La funzione parse\_cmd analizza gli argomenti della riga di comando per configurare le variabili globali.

Complessità di Shell Sort:

-Nel **caso migliore**: O(n log n) quando la sequenza dei gap è scelta in modo ottimale;

-Nel **caso medio**: O(n log2 n);

-Il **caso peggiore** è uguale al caso medio.

Operazioni di lettura e scrittura:

-La lettura dell'array dal file richiede **O(n)** operazioni;

-Ogni test esegue Shell Sort su un array di dimensione n, con **ntests** test, quindi, la complessità totale per tutte le operazioni di lettura e ordinamento è **O(ntests ⋅ n log2 n).**

Operazioni di input/output:

Leggere i dati dal file e scrivere i risultati nel file di output richiede operazioni lineari in **O(n)** per ciascun test, quindi, la complessità totale è **O(ntests⋅n).**

Riassunto, il programma esegue **ntests** ordinamenti con Shell Sort, misurando e riportando il numero di letture effettuate, con una complessità totale dominata da **O(ntests ⋅ nlog2 n)**.

**Laboratorio 2 – Albero bilanciato e SubAlbero**

Il codice fornisce le funzioni per la gestione di alberi binari come la creazione, traversamenti, bilanciamento e confronto di alberi.

Struttura dell'albero binario:

-Definisce la struttura `node\_t`, che rappresenta un nodo dell'albero binario con un valore (`val`) e puntatori ai figli sinistro (`L`) e destro (`R`);

-Possiede funzioni per la creazione di nuovi nodi e l'inserimento di figli sinistro e destro.

**La complessità** di creazione di un nodo con `node\_new` o l’inserimento di un figlio L o R, è di **O(1)**.

Inserimento casuale di nodi:

-La funzione `insert\_random\_rec` inserisce nodi casualmente, limitando il numero massimo di nodi tramite `max\_nodes` per evitare alberi troppo grandi;

-Questa funzione ha una **complessità** **media di O(n)**, dove **n** è il numero di nodi da inserire, perché ogni nodo può essere visitato una volta durante l'inserimento.

Traversamenti dell'albero:

-Le funzioni `inOrder`, `preOrder`, `postOrder` e `EulerOrder` eseguono dei traversamenti ricorsivi;

-La **complessità** di ciascuna di queste funzioni è **O(n)**, dove **n** è il numero di nodi nell'albero, perché ogni nodo viene visitato esattamente una volta.

Bilanciamento dell'albero:

- La funzione `height` calcola l'altezza dell'albero in **O(n)** con n il numero di nodi nel sottoalbero;

- La funzione `isBalanced` verifica se l'albero è bilanciato, chiamando `height` su ciascun nodo. La **complessità** **complessiva** **è pari a O(n2)**, in quanto ogni nodo calcola l'altezza dei suoi sottoalberi.

Confronto di alberi:

- La funzione `isEqual` verifica se due alberi sono identici, confrontando i valori dei nodi e chiamandosi ricorsivamente sui figli. Ha una **complessità pari a O(n)** con n il numero di nodi del sottoalbero;

- La funzione `isSubtree` verifica se un albero è un sottoalbero di un altro, chiamando `isEqual` su ogni nodo dell'albero principale. Ha una **complessità pari a O(n \* m)** con n il numero di nodi dell’albero principale ed m il numero di nodi del sottoalbero;

**Laboratorio 3 – Grafo visitato e lunghezza ciclo più lungo**

Il codice in questione riguarda la rappresentazione e la visita di un grafo utilizzando una lista di adiacenza, con particolare enfasi sulla ricerca in profondità (DFS, Depth-First Search).

Definizioni e Dichiarazioni:

-Variabili globali: Il codice definisce diverse variabili globali utilizzate per tracciare operazioni, dimensioni, contatori, ecc.

Strutture dati:

-`node\_t`: Definisce un nodo per una lista collegata, contenente un valore e un puntatore al prossimo nodo;

- `list\_t`: Definisce una lista collegata, contenente un puntatore alla testa della lista;

- `NodeInfo`: Struttura per tracciare i nodi visitati e i loro livelli durante la DFS.

Grafo:

- `V`: Array dei valori dei nodi;

- `V\_visitato`: Array per tracciare se un nodo è stato visitato;

- `E`: Array di puntatori a liste di adiacenza.

Funzioni utalitarie:

- list\_new: Crea e inizializza una nuova lista;

- list\_insert\_front: Inserisce un nodo all'inizio di una lista;

- list\_print: Stampa il contenuto di una lista;

- node\_print: Stampa un nodo e i suoi archi;

- graph\_print: Stampa l'intero grafo;

- print\_array e print\_array\_graph: Funzioni per stampare array e rappresentarli graficamente.

Main:

Inizializzazione del grafo:

-Allocazione degli array `V` e `V\_visitato`;

-Creazione e inizializzazione delle liste di adiacenza per ogni nodo;

-Popolazione delle liste di adiacenza con archi casuali;

-Stampo il grafo attraverso la funzione `graph\_print` per visualizzare il grafo;

DFS (Depth-First Search):

-La DFS viene eseguita per esplorare il grafo e calcolare la lunghezza del ciclo più lungo;

-DFS(int n, NodeInfo\* nodeInfo, int parent, int& longestCycle)`: Questa funzione DFS ricorsiva visita il grafo e calcola la lunghezza del ciclo più lungo.

Visita di grafi disconnessi:

-`visitDisconnectedGraph`: Questa funzione gestisce il caso di grafi disconnessi avviando una DFS da ogni nodo non visitato.

Output:

-Stampa la lunghezza del ciclo più lungo trovato.

**Complessità:**

-Inizializzazione: La **complessità di inizializzazione** degli array e delle liste di adiacenza **è O(n + m)**, dove n è il numero di nodi e m è il numero di archi.

-Inserimento in lista: Ogni **inserimento** in una lista di adiacenza **ha complessità O(1)**, quindi la complessità totale è pari a **O(m)**.

-La **complessità della DFS è O(n + m)**, poiché visita ogni nodo e ogni arco esattamente una volta.

-La **visita di grafi disconnessi ha complessità** **O(n + m)** poiché visita ogni componente connessa separatamente.

-Il calcolo della **lunghezza del ciclo più lungo** è integrata nella DFS, quindi **non aggiunge ulteriore complessità asintotica**.

**La complessità totale del programma**, considerando tutte le operazioni di inizializzazione, costruzione del grafo ed esecuzione della DFS, **è O(n + m)**, dove n è il numero di nodi e m è il numero di archi nel grafo.

**Laboratorio 4 – Shortest Path e Bellman-Ford algorithm**

Il programma implementa la costruzione e gestione di un grafo orientato, rappresentato tramite liste di adiacenza, e fornisce algoritmi per il calcolo del percorso minimo come Dijkstra e Bellman-Ford. Analizziamo i vari componenti e la loro complessità.

Definizione delle Strutture:

-`Node`: rappresenta un nodo dell'albero con valore `val`, peso `w`, e puntatore al nodo successivo `next`;

-`List`: rappresenta una lista collegata di nodi con un puntatore alla testa `head`;

-`nodeData`: rappresenta dati relativi a un nodo del grafo con i campi `prev`, `dist`, `visitato`, `nodeNumber`.

Funzioni di Utilità:

-`print\_node`: Stampa le informazioni di un nodo specifico in un file di output;

-`print\_graph`: Stampa l'intero grafo chiamando `print\_node` per ogni nodo;

-`crea\_lista`: Crea e inizializza una nuova lista di adiacenza;

-`print\_list`: Stampa gli elementi di una lista di adiacenza;

-`inserisci\_in\_testa`: Inserisce un nuovo nodo all'inizio di una lista di adiacenza.

Heap Minimo:

-`swap`, `adjustIndex`, `minHeapify`, `decreaseDistance`, `minHeapRootDelete`: Funzioni per la gestione di un heap minimo, usato per implementare efficientemente la struttura dati per l'algoritmo di Dijkstra.

Algoritmi di Percorso Minimo:

-`shortest\_path`: Implementa l'algoritmo di Dijkstra per trovare il percorso più breve da un nodo sorgente a tutti gli altri nodi nel grafo.

-`bellmanFord`: Implementa l'algoritmo di Bellman-Ford, che gestisce anche i grafi con archi di peso negativo.

**Complessità:**

-**`print\_node` e `print\_graph` hanno complessità pari a O(n + m)** dove n è il numero di nodi e m è il numero di archi, poiché stampa tutti i nodi e gli archi del grafo;

-**crea\_lista` e `inserisci\_in\_testa` hanno complessità pari a O(1)**, poiché la creazione della lista e l'inserimento di un nodo in testa sono operazioni costanti;

**-minHeapify` e `decreaseDistance` hanno complessità pari a O(log n)** per chiamata, dove n è il numero di nodi;

**-minHeapRootDelete` ha complessità totale pari a O(log n)**, poiché richiede una operazione di `minHeapify`.

**-shortest\_path` (Dijkstra) ha complessità pari a O((n + m) log n)**, dove n è il numero di nodi ed m è il numero di archi. La complessità logaritmica deriva dall'uso del heap minimo per ottenere il nodo con la distanza minima e aggiornare le distanze.

**-`BellmanFord` ha una complessità pari a O(n \* m)**, dove n è il numero di nodi e m è il numero di archi. Questo algoritmo attraversa tutti gli archi per ogni nodo.

**Laboratorio 5 – Prim algorithm**

Il programma in questione implementa l'algoritmo di Prim per trovare l'albero di copertura minimo (MST, Minimum Spanning Tree) di un grafo non orientato e pesato.

-StrutturaEdge: rappresenta un arco del grafo, dove V è il vertice di destinazione e weight è il peso dell'arco;

-FunzionecreateGraph: Crea un grafo con n vertici rappresentato come un vettore di vettori di archi;

**-**FunzioneaddEdge: Aggiunge un arco non orientato tra i vertici U e V con un determinato peso;

**-**FunzioneprimMST: Implementa l'algoritmo di Prim per trovare l'MST a partire dal vertice start;

Utilizza una coda di priorità per selezionare l'arco di peso minimo;

Inizializza le chiavi dei vertici a infinito (eccetto il vertice iniziale) e una coda di priorità;

Processa i vertici, aggiornando le chiavi e i genitori e aggiungendo gli archi MST al risultato.

-FunzioneprintMST: Stampa gli archi dell'MST;

-Main: Il programma principale richiede all'utente il numero di vertici e archi, crea il grafo, aggiunge gli archi, calcola l'MST e lo stampa.

**Complessità:**

-**createGraph ha complessità pari a O(V)**, dove n è il numero di vertici;

-**addEdge: ha complessità pari a O(1) per ogni chiamata**;

-primMST: Considerando una coda di priorità binaria (heap binario), **l'algoritmo di Prim ha complessità pari a O((E + V) log V)**, dove E è il numero di archi e V è il numero di vertici;

-**pq.push e pq.pop hanno complessità pari a O(log V)**;

-**L’estrazione dei vertici con chiave minima ha complessità pari a O(V log V)** con V il numero di vertici;

-**Aggiornamento delle Chiavi dei Vertici Adiacenti** **ha complessità pari a O(E log V)** con E il numero di archi.

**Laboratorio 6 – SubSequence e Score di due codici sorgente**

Il programma in questione è un tool di confronto testuale che implementa due algoritmi principali: il calcolo della "Longest Common Subsequence" (LCS) e un allineamento di sequenze, oltre a contenere delle funzioni per la gestione di grafi e liste collegate. La sua complessità complessiva è dominata dalle operazioni di confronto e allineamento delle stringhe, oltre che dalla gestione dei grafi per la parte di SCC.

-La funzione `readFileIntoString` legge il contenuto di un file e lo restituisce come stringa;

-Il main legge due file ("file1.txt" e "file2.txt"), visualizza i loro contenuti, calcola la LCS e poi allinea le due sequenze;

-La funzione `longestCommonSubsequence` calcola la sottosequenza comune più lunga tra due stringhe usando la programmazione dinamica. **Ha complessità temporale pari a O(m \* n)**, dove m e n sono le lunghezze delle due stringhe. **Ha complessità spaziale pari a O(m \* n)**, dovuta alla matrice utilizzata per memorizzare i risultati intermedi;

-La funzione `allinea` allinea due stringhe con penalità per le operazioni di match, mismatch e inserimento/eliminazione. **Ha complessità temporale pari a O(s1 \* s2)**, dove s1 e s2 sono le lunghezze delle due stringhe. **Ha complessità spaziale pari a O(s1 \* s2)**, dovuta alla matrice utilizzata per il calcolo dell'allineamento;

Strutture dati:

-Lista collegata: Implementata tramite le strutture `node` e `list`, con funzioni per la creazione, inserimento e cancellazione di nodi.

-Stack: Implementato come una lista collegata, con operazioni di push, pop e top.

-Grafo: Rappresentato tramite una lista di adiacenza, con funzioni per l'aggiunta di nodi e archi, e per la stampa del grafo.

SCC (Strongly Connected Components):

-La funzione `scc` implementa l'algoritmo di Tarjan per trovare le componenti fortemente connesse in un grafo. **Ha complessità temporale pari a O(V + E)**, dove V è il numero di nodi ed E è il numero di archi nel grafo. **Ha complessità spaziale pari a O(V)**, per memorizzare gli stati dei nodi e lo stack.

FIB:

La funzione calcola i numeri di Fibonacci con memorizzazione. **Ha complessità temporale pari a O(n)**, dove n è l'input per la funzione `fib`. **Ha complessità spaziale pari a O(n)**, per memorizzare i valori calcolati di Fibonacci.