Laboratorio Di Algoritmi e Strutture Dati Anno 2021/2022

Stefano Peddoni Matr 839138

Davide Laguardia Matr 822358

Relazione Esercizio 1

Nell'Esercizio 1 è stata implementata una libreria generica con alcune funzioni:

```c
/* Create new Generic Array */
GenericArray* GenericArray_create(GenericArrayCmp);

/* Returns:
 * - 1 if the array is empty
 * - 0 if the array is not empty
 */
int GenericArray_empty(GenericArray* generic_array);

/* Returns the capacity of the Generic Array */
int GenericArray_capacity(GenericArray* generic_array);

/* Returns the number of elements currently stored in the Generic Array */
unsigned long GenericArray_size(GenericArray* generic_array);

/* Deallocates the Generic Array */
void GenericArray_free(GenericArray* generic_array);

/* It accepts as input a pointer to an Generic Array and an integer "i"
   and it returns the pointer to the i-th element of the Generic Array */
void* GenericArray_get(GenericArray* generic_array, int i);

/* Insert an element inside the Generic Array */
void GenericArray_insert(GenericArray* generic_array, void* element);

/* Method that doubles the size of the Generic Array */
void GenericArray_check_and_realloc(GenericArray* generic_array);

/* Reverses the order of elements in the Generic Array */
void GenericArray_reverse(GenericArray* generic_array);
```

Dopo aver creato la libreria per il generic_array, e testato le funzioni create con alcuni test; abbiamo creato l'applicativo per poter eseguire i due algoritmi richiesti: Insertion Sort (utilizzando la ricerca binaria) e Quick Sort, ordinando i dati che sono presenti all'interno del file records in maniera crescente.

I due algoritmi presi in questione sono caratterizzati in questa maniera:

ALGORITMO	CASO PEGGIORE	CASO MIGLIORE
Quick Sort	O(n^2)	O(nlogn)
InsertionSort	O(n^2)	O(n)

Di seguito abbiamo calcolato per ogni field (int, string e float) i tempi di ordinamento per ciascun algoritmo. Di seguito alcuni esempi di dati che ho ottenuto:

Nell'applicativo sono stati implementati i due algoritmi che abbiamo citato sopra nella tabella. Abbiamo effettuato diverse verifiche con file di dimensioni diverse ed abbiamo ottenuto questi risultati:

RECORDS (STRING)	QUICK SORT	INSERTION SORT
100.000	0.30s	11.34s
500.000	1.69s	104.92s
700.000	2.46s	233.90s
20.0000	87.32s	∞

RECORDS (INT)	QUICK SORT	INSERTION SORT
100.000	0.28s	19.12s
500.000	1.51s	109.47s
700.000	2.19s	220.26s
20.0000	78.55s	∞

RECORDS (FLOAT)	QUICK SORT	INSERTION SORT
100.000	0.30s	17.12s
500.000	1.52s	142.20s
700.000	2.17s	225.60
20.0000	78.77s	∞

Insertion Sort

Possiamo quindi notare che i file contenenti sempre più records, richiedono un tempo maggiore, e nel caso dei 20.000.000 di records il tempo per l'ordinamento risulta essere infinito. Questo dipende anche dalla macchina con cui si lavora, se si utilizza una macchina virtuale i tempi saranno maggiori.

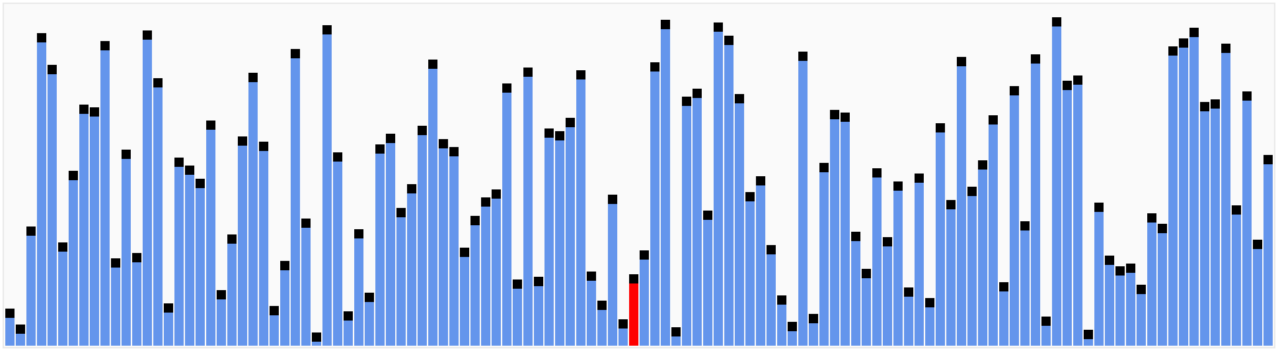
Quick Sort

Possiamo notare che questo algoritmo riesce ad ordinare in maniera crescente i records fino ai 20.000.000. Il field "string" ci mette più tempo rispetto a "int" e "float", ma, nonostante ciò, termina l'ordinamento in maniera positiva.
All'interno dell'algoritmo del Quick Sort viene utilizzato il pivot per la scelta dell'elemento.

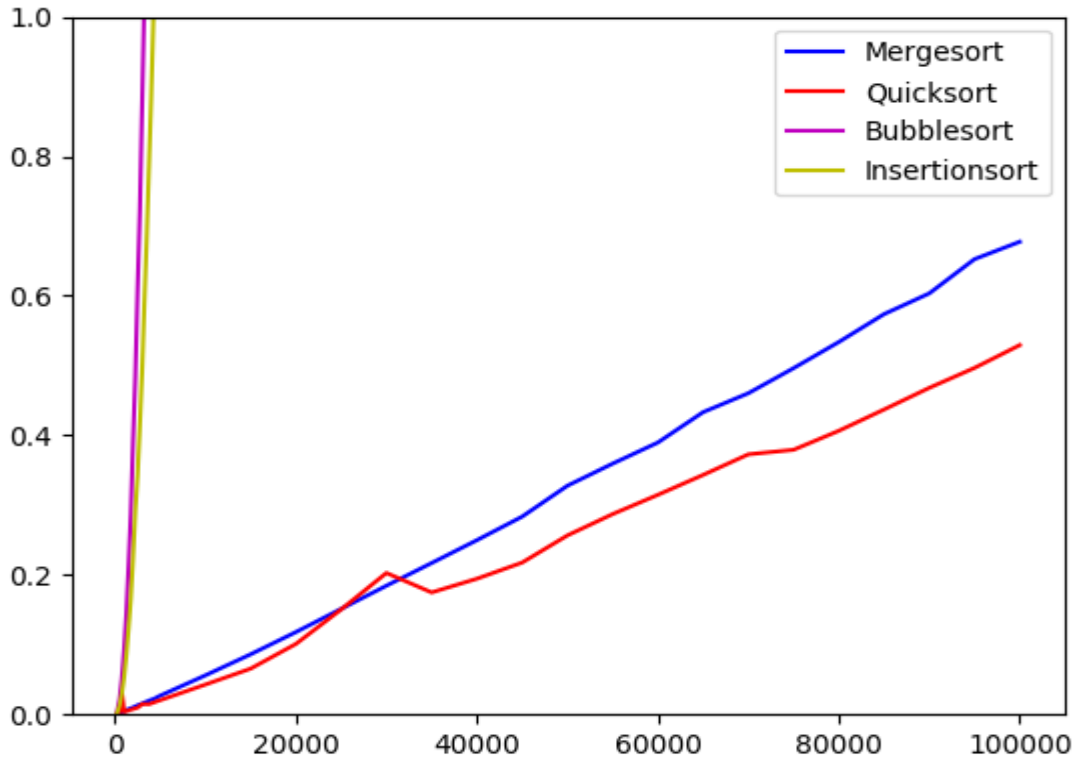
Scelta del Pivot

La scelta del pivot nel Quick Sort può avvenire in diversi casi:

- Primo elemento: in questo caso viene preso il valore più basso (low), quindi il primo elemento della lista.
- Ultimo elemento: in questo caso viene preso il valore più alto (high), quindi l'ultimo elemento della lista.
- Elemento centrale: in questo caso è la soluzione migliore, poiché con records elevati riesce ad essere più performante a livello di tempistica.



Di seguito troviamo un grafico che rappresenta i tempi con il quale l'algoritmo riesce ad ordinare un numero di elementi. Possiamo subito notare quello che è accaduto a noi durante lo svolgimento dell'esercizio; l'algoritmo Insertion Sort ci mette molto più tempo rispetto al Quick Sort.



Laboratorio Di Algoritmi e Strutture Dati Anno 2021/2022

Stefano Peddoni Matr 839138

Davide Laguardia Matr 822358

Relazione Esercizio 2

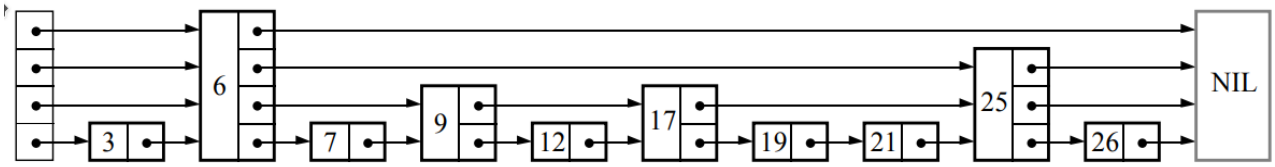
Frase:

Quando avevo cinqve anni, mia made mi perpeteva sempre che la felicita e la chiave della vita. Quando andai a squola mi domandrono come vuolessi essere da grande. Io scrissi: selice. Mi dissero che non avevo capito il corpito, e io dissi loro che non avevano capito la wita.

SkipList:

Una skiplist è una struttura dati probabilistica per la memorizzazione di una lista ordinata di elementi. Al contrario delle liste concatenate classiche, la skiplist è una struttura dati probabilistica che permette di realizzare l'operazione di ricerca con complessità O(log n) in termini di tempo. Anche le operazioni di inserimento e cancellazione di elementi possono essere realizzate in tempo O(log n). Per questa ragione, la skiplist è una delle strutture dati che vengono spesso utilizzate per indicizzare dati.

All'interno della skiplist è possibile utilizzare un valore MAX_HEIGHT (è una costante che definisce il massimo numero di puntatori che possono esserci in un singolo nodo della skiplist. Come si vede nella figura, ogni nodo può avere un numero distinto di puntatori.



Verificando i valori di MAX_HEIGHT possiamo notare che per un valore < di 10 i tempi di ricerca con la skiplist aumentano; si consiglia quindi di utilizzare un valore superiore a 10 per riscontrare dei tempi minori e accettabili.

Laboratorio Di Algoritmi e Strutture Dati Anno 2021/2022

Stefano Peddoni Matr 839138

Davide Laguardia Matr 822358

Relazione Esercizio 3 – 4

Nell'esercizio 3 è stato implementato un Heap minimo che verrà poi utilizzato all'interno di Dijkstra.

Nell'esercizio 4 è stata implementata la struttura di un Grafo.

- Qui di seguito elencheremo i metodi usati dall'Hashtable all'interno di Graph:
    - Private Hashtable<T, Vertex<T,S>> hash;

METODO	COMPLESSITA'	DESCRIZIONE
hash.put	O(1)	Viene utilizzata per inserire la coppia <chiave, valore>.
hash.containsKey	O(1)	Viene utilizzata per verificare se la chiave è contenuta nel grafo e ritorna se è presente
hash.get	O(1)	Viene utilizzata per verificare la chiave e restituisce il valore associato.
hash.size	O(1)	Restituisce la dimensione del grafo
hash.remove	O(1)	Viene utilizzata per rimuovere l'elemento dal grafo
hash.keySet	O(1)	Restituisce una vista delle chiavi

Inoltre, è stata utilizzata anche un'interfaccia Set: Set in Java **è un'interfaccia che fa parte di Java Collection Framework e implementa l'interfaccia Collection**. Una raccolta di set fornisce le caratteristiche di un set matematico. Un set può essere definito come una raccolta di oggetti non ordinati e non può contenere valori duplicati. E' stato quindi scelto di utilizzare la Set, al posto della List proprio per il fatto che la Set viene definita come una raccolta di oggetti senza duplicati, cosa che invece non permette List.

Le seguenti operazioni richieste sono:

Creazione di un grafo vuoto: In **graph**, è stato creato un grafo vuoto attraverso una Hashtable, un parametro direct che essendo boolean determina se il grafo menzionato è diretto (TRUE) oppure non diretto (FALSE). La complessità del problema è di O(1) non avendo cicli.

Aggiunta di un nodo: In **addVertex**, aggiungo un nodo all'interno del grafo, ritorno TRUE se il parametro v viene aggiunto al grafo, FALSE nel caso contrario. La complessità asintotica del problema è di O(1) non avendo cicli.

Aggiunta di un arco: In **addEdge**, aggiungo un arco all'interno del grafo. In questo caso nel metodo addEdge vado a verificare se il grafo è orientato o non orientato. La complessità del problema è di O(1) non avendo cicli.

Verifica se il grafo è diretto: In **isDirect**, verifico se il grafo in questione è diretto o no. Nel primo caso ritorna TRUE, nel secondo caso FALSE. La complessità del problema è di O(1) non avendo cicli.

Verifica se il grafo è indiretto: In **isUndirect**, verifico se il grafo in questione è indiretto o no. Nel primo caso ritorna TRUE, nel secondo caso FALSE. La complessità del problema è di O(1) non avendo cicli.

Verifica se il grafo contiene un dato nodo: In **containsVertex**, verifico se un dato nodo è contenuto all'interno del grafo. In questo caso verifico se vertexToFind è contenuto all'interno di esso. La complessità del problema è di O(1) non avendo cicli.

Verifica se il grafo contiene un dato arco: In **containsEdge**, verifico se un dato arco è contenuto all'interno del grafo. In questo caso verifico se source e destination sono contenuti all'interno di esso. La complessità del problema è di O(1) non avendo cicli.

Cancellazione di un nodo: In **removeVertex**, passiamo come parametro v l'elemento che vogliamo cancellare. La complessità asintotica del problema è O(n) poiché all'interno del metodo abbiamo un ciclo for.

Cancellazione di un arco: In **removeEdge**, passiamo come parametro source e destination gli elementi che vogliamo cancellare. La complessità del problema è di O(1) non avendo cicli.

Determinazione del numero di nodi: In **getNumberVertex** determino il numero di nodi presenti nel grafo. La complessità asintotica è O(1) non avendo cicli.

Determinazione del numero di archi: In **getNumerEdges** determino il numero di archi presenti nel grafo. La complessità asintotica del problema è di O(n) avendo un ciclo while all'interno.

Recupero dei nodi del grafo: In questo caso abbiamo creato due funzioni che permettono di recuperare la sorgente del vertice con **getVertex** e **getAllVertex** per recuperare tutti i nodi presenti nel grafo. La complessità asintotica del problema è O(n) avendo un ciclo for all'interno.

Recupero degli archi del grafo: In questo caso abbiamo creato due funzioni chiamate **getEdge** prendendo in considerazione un singolo arco e **getEdges** per recuperare tutti gli archi presenti nel grafo. La complessità asintotica del problema è O(n) avendo un ciclo for all'interno.

Recupero nodi adiacenti di un dato nodo: In **getAdjacentVertex**, recupero tutti i nodi adiacenti presenti nel grafo. La complessità asintotica del problema è O(1).

Recupero etichetta associata a una coppia di nodi: Utilizziamo **getVertexLabel** e **getAdjacentVertexLabel**, recupero l'etichetta a,b associata ad una coppia di nodi presenti nel grafo. La complessità asintotica del problema è O(1).

Implementazione di Dijkstra:

L'algoritmo di Dijkstra è un algoritmo per trovare i percorsi più brevi tra i nodi in un grafo. Per un dato nodo sorgente nel graph, l'algoritmo trova il percorso più breve tra quel nodo e ogni altro nodo. L'algoritmo di Dijkstra si basa sul principio del rilassamento, in cui valori più accurati sostituiscono gradualmente un'approssimazione alla distanza corretta fino a raggiungere la distanza più breve.

Partendo dalla libreria creata all'interno di Graph e all'interno di Heap e dalle nozioni acquisite dal corso di teoria, abbiamo creato una classe separata per l'implementazione dell'algoritmo di Dijkstra, e convertito lo pseudo codice visto in codice Java.

Dopo aver implementato l'algoritmo di Dijkstra, abbiamo creato una classe DijkstraMain contente l'avvio dell'algoritmo per verificare il percorso tra Torino e Catania tramite il file csv.

Per poter avviare sia l'esercizio 3 che l'esercizio 4 è stato creato un file readme all'interno delle due cartelle con la spiegazione per l'esecuzione del programma contenente i test.

Risultati: Come sorgente si è scelta Torino, mentre come destinazione si è scelta Catania; il risultato del percorso tra le due città risulta essere di 1207.68Km