UNIVERSITÀ DEGLI STUDI DI TORINO

Progetto di Algoritmi e Strutture Dati a.a. 2023/2024

Tommasi Samuele

Indice

1 Esercizio uno

Questo programma C consente di ordinare un file **.csv* specificato dall'utente tramite linea di comando. L'utente può scegliere quale campo ordinare e quale algoritmo di ordinamento utilizzare attraverso i parametri di esecuzione. Il programma legge i dati dal file, li ordina in base al campo scelto utilizzando l'algoritmo selezionato, quindi scrive i dati ordinati su un nuovo file **.csv* di output, anch'esso specificato dall'utente.

1.1 Struttura di esecuzione del programma

La compilazione viene effettuata tramite **Makefile** (utilizzare il comando *make help* per maggiori informazioni); successivamente, per eseguire il programma, è necessario immettere il seguente comando nel terminale:

bin/main_ex1	file_input.csv	file_output.csv	field	algorithm

- file_input.csv: il file da ordinare.
- file_output.csv: il file dove verranno salvati i dati ordinati
- field: il campo del file **.csv* da ordinare.
- algorithm: l'algoritmo di ordinamento da utilizzare.

1.2 MergeSort e QuickSort

L'esercizio proposto imponeva di implementare i seguenti algoritmi di ordinamento:

1.2.1 Implementazione del QuickSort

Ho implementato l'algoritmo secondo lo schema base del QuickSort, scegliendo il pivot in maniera non casuale e utilizzando un approccio ricorsivo.

Passi:

- **Divide**: scelta del pivot come ultimo elemento dell'array e partizionamento di Lomuto intorno ad esso.
- **Impera**: applicazione ricorsiva dell'algoritmo alle due partizioni.

1.2.2 Implementazione del MergeSort

Ho implementato l'algoritmo utilizzando la tecnica Top-Down del MergeSort, ovvero suddividendo ricorsivamente l'array in sotto-array più piccoli fino a che non arrivano a contenere un solo elemento per poi combinarli in maniera ordinata tramite la funzione merge.

Passi:

- **Divide**: divisione dell'array in due sotto-array.
- **Impera**: ordinamento dei due sotto-array ricorsivamente.
- **Merge**: fusione dei due sotto-array in un unico array ordinato.

1.3 Struttura dei file di I/O

Il corretto funzionamento dei due algoritmi e le loro prestazioni sono testati sul file *records.csv* contenente 20 milioni di record da ordinare, organizzati nel seguente modo:

- id: identificatore univoco del record
- field1: elemento di tipo Int
- field2: elemento di tipo String
- field3: elemento di tipo Floating Point

Affinché il programma potesse operare su ciascuno di questi campi, gli algoritmi di ordinamento sono stati implementati utilizzando i tipi generici e per ciascuno di essi sono stati implementati metodi di confronto di conseguenza, in questo caso string, int e float.

Le variazioni nell'input dell'utente, riguardanti il campo selezionato e l'algoritmo che si intende utilizzare, influenzano i tempi di esecuzione del programma, presentando tre diversi scenari per ciascun algoritmo.

1.4 Aspettative

Vista l'implementazione di QuickSort nel programma prevedo tempi di esecuzioni relativamente più variabili rispetto al MergeSort, in entrambi i casi un possibile rallentamento è fondamentalmente legato al tipo di dato da ordinare, di fatto le stringhe in C non sono altro che puntatori a char, perció il loro confronto costoso potrebbe provocare dei rallentamenti, in particolare al QuickSort vista la possibilità di imbattersi nel caso peggiore $O(n^2)$, mentre per quanto riguarda gli interi e floating point, visto il loro confronto decisamente più rapido, i tempi di esecuzioni potrebbero essere simili. Un altro motivo di rallentamento sarà sicuramente il processo di copia dell'array ordinato all'interno del file di output.

1.4.1 Possibili debolezze del QuickSort

La scelta del pivot non è ottima poiché l'algoritmo sceglie sempre l'ultimo elemento dell'array, questo potrebbe generare partizioni sbilanciate che porteranno a un numero elevato di ricorsioni e di conseguenza rallentare notevolmente il programma. Per migliorare questo aspetto si potrebbero utilizzare il **QuickSort median-of-three** (selezionare un pivot che sia vicino alla mediana degli elementi) o il **randomized QuickSort** (selezione casuale del pivot).

La profondità della ricorsione può diventare problematica quando si lavora con dataset molto grandi (nel nostro caso 20 milioni di stringhe), perché ogni chiamata ricorsiva aggiunge un frame di stack alla pila delle chiamate e se la profondità diventa troppo grande possiamo incorrere in un problema di StackOverflow. Per risolvere questo problema si potrebbe eseguire un controllo in testa all'algoritmo per controllare la profondità massima della ricorsione e passare a un algoritmo non ricorsivo come InsertionSort (**qsort** function in C).

L'accesso ai dati attraverso il QuickSort avviene spesso in maniera non sequenziale, questo perché durante il partizionamento gli elementi dell'array vengono confrontati e scambiati in base al pivot.

1.4.2 Possibili debolezze del MergeSort

A differenza del QuickSort il MergeSort presenta meno debolezze e dovrebbe risultare molto efficiente nonostante utilizzi memoria aggiuntiva per gli array temporanei (QuickSort è un algoritmo *in-place*). Ciò deriva dal fatto che

l'algoritmo divide sempre a metà l'array questo porta a una profondità di ricorsione minore, impedendo una degradazione temporale come $O(n^2)$ che si presenta invece in caso di pivot scomodo nel QuickSort, anche se non si evade completamente la possibilità di StackOverflow.

In sintesi, ogni debolezza indicata è solamente un'ipotesi, non è detto che gli algoritmi subiscono rallentamenti durante il processo di ordinamento, ma nel caso ciò dovesse accadere le conseguenze potrebbero essere tra quelle indicate precedentemente.

1.5 Test e Conclusioni

	Integer	String	Floating Point
MergeSort	29.865s	29.317s	29.940s
QuickSort	33.140s	**FAILED**	31.801s

Tutti i test sono finiti più o meno come previsto ad eccezione del fallimento verificatosi nell'ordinamento del campo string tramite QuickSort, le motivazioni come suggerito in precedenza potrebbero essere legate ad una cattiva scelta per quanto riguarda il pivot.

Evidentemente il dataset non è adeguato per l'ordinamento tramite QuickSort, tuttavia l'algoritmo può essere migliorato, per esempio, tramite una miglioria nella scelta del pivot o applicando un approccio diverso: **randomized QuickSort**, **three-way QuickSort** o **QuickSort median-of-three**.

2 Esercizio due

Questo programma C implementa un'applicazione che utilizza una delle varianti dell'algoritmo che calcola l'**edit distance** per determinare, per ogni parola *w* in un file **.txt* specificato dall'utente una breve lista di parole in *dictionary.txt* con edit distance minima da *w*.

2.1 Struttura di esecuzione del programma

Analogalmente all'Esercizio uno anche nel due la compilazione viene effettuata tramite **Makefile** (*make help* per maggiori informazioni). Per eseguire il programma da terminale è necessario inserire il seguente comando:

bin/main_ex2	dictionary.txt	correctme.txt

- dictionary.txt: file dizionario contenente tutte le parole corrette.
- correctme.txt: file da correggere.

2.2 Edit Distance

L'edit distance è una misura della differenza tra due stringhe, ovvero il numero minimo di operazioni necessarie per trasformare una stringa in un altra. Tale distanza può essere calcolata in modi differenti a seconda dell'algoritmo utilizzato.

Quello implementato dal programma è una variante dell'**algoritmo di Levenshtein**, che normalmente permetterebbe di calcolare l'edit distance con tre operazioni: inserimento, cancellazione e sostituzione, ma nel nostro caso permette solamente le prime due, pertanto la sostituzione sarà rappresentata come una cancellazione più un inserimento.

2.2.1 Implementazione

L'esercizio proposto imponeva di implementare l'algoritmo due volte con approcci differenti: il primo approccio, suggerito dalla consegna, era di tipo prettamente ricorsivo senza richieste particolari, mentre il secondo richiedeva

di implementare lo stesso algoritmo utilizzando una strategia di **programmazione dinamica**, mantenendo però la ricorsione.

Per implementare tale algoritmo mi sono basato sulla struttura della funzione offerta dalla consegna e, dal momento in cui l'esercizio richiedeva di mantenere la ricorsione, ho adottato l'approccio **Top-Down** o **Memoization** utilizzando una funzione "helper" d'appoggio.

Anche se non necessario ho implementato anche la funzione utilizzando l'approccio completamente iterativo o **Bottom-up**

2.3 Test e Conclusioni

In questo esercizio ho riportato solamente i risultati portati dall'esecuzione del programma con la funzione che adotta la strategia di programmazione dinamica, poiché la soluzione ricorsiva, per quanto corretta, risultava troppo lenta a causa dell'eccessivo numero di confronti necessari per stabilire una e.d. minima per tutte le 48 parole del file correctme.txt, che andavano di fatto confrontate con tutte le oltre 660 mila parole del dizionario.

	Top-Down	Bottom-Up
correctme.txt	18.578s	11.920s

Il programma funziona in maniera matematicamente perfetta, terminando correttamente con entrambe le soluzioni adottate (per verificare l'output eseguire da linea di comando).

Come ci si aspetta l'approccio Bottom-Up è più efficiente poiché non deve gestire la ricorsione e può dunque evitare il sovraccarico, in termini di stack, delle chiamate.

2.4 Riflessioni

Come ho detto, il programma funziona alla perfezione per quanto riguarda la parte matematica, ovvero il calcolo della distanza di edit, ma non sempre la soluzione con la distanza di edit minore è quella corretta. Poiché non abbiamo i mezzi per controllare il contesto a cui la parola da correggere fa riferimento, è possibile incorrere in situazioni in cui la correzione proposta, sebbene abbia una distanza di edit minore, non ha senso.

Ecco alcuni esempi ricavati direttamente dal test su correctme.txt:

- **selice** non viene corretta con felice poiché la parola esiste nel dizionario (e.d. = 0).
- **squola** non viene corretta con scuola (e.d. = 2) ma con suola, che di fatto ha edit distance minore.
- **made** non viene corretta con madre poiché la parola esiste nel dizionario.

Per risolvere questo problema, una possibile soluzione potrebbe essere quella di individuare tutte le parole con una distanza di edit di uno o due rispetto al minimo. Tuttavia, ciò comporterebbe una stampa eccessiva, andando contro le indicazioni dell'esercizio.