Algoritmi - Relazione di Laboratorio

Di Lucchio Matteo

Matricola 795310

Email matteo.dilucchio@edu.unito.it

Anno accademico 2017/18

Esercizio 1 - parte 1

Lo scopo dell'esercizio 1 era di realizzare due librerie che permettessero di ordinare in ordine crescente o decrescente gli interi letti dal file *integers.csv* usando gli algoritmi **insertion sort** e **merge sort**.

Insertion Sort

Considerando un vettore di interi, Insertion Sort e' un algoritmo di ordinamento in cui la parte sinistra del vettore e' ordinata (vero per vettori da un solo elemento) e ogni nuovo inserimento dalla parte destra alla parte sinistra mantiene il vettore ordinato. Ad ogni iterazione nel caso peggiore si scorre tutta la parte ordinata fino all'ultimo indice, nel caso migliore invece ci si blocca al primo elemento.
Questo algoritmo e' quindi ottimo per vettori gia' quasi ordinati.

Vista la mole di dati da analizzare ho deciso di modificare la normale implementazione di questo algoritmo per tentarne una versione ottimizzata di tipo *DIVIDE ET IMPERA*, in cui la ricerca dell'indice in cui inserire l'elemento analizzato avviene dimezzando di volta in volta l'area considerata.

Analisi della complessita'di Insertion Sort iterativo

Caso ottimo $T_{\mathrm{cm}}(n) \approx an$

Caso peggiore $T_{\mathrm{cp}}(n) \approx an^2$

Analisi della complessita'di Insertion Sort *Divide Et Impera*

$$T(n) \in \Theta(n \log n)$$

Fallimento dell'algoritmo

Nonostante i test positivi e il miglioramento delle prestazioni rispetto al primo approccio (Insertion Sort iterativo), *Divide et Impera* non e' stato fruttuoso, infatti dopo 10 minuti l'algoritmo aveva analizzato circa il 3% dei dati.

Merge Sort

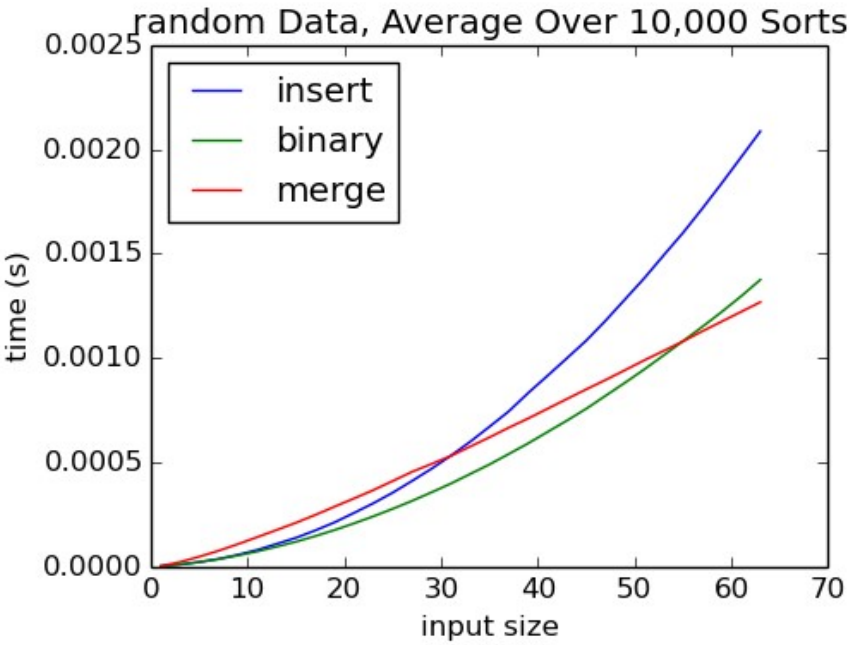
Merge Sort e' un algoritmo di ricerca a partizioni bilanciate che opera per fusione.
L'idea alla base dell'algoritmo e' di spezzettare il problema in sottoproblemi fino ad avere un sottoproblema facilmente risolvibile, per poi fondere la soluzione al sottoproblema precedente.

Anche in questo caso la complessita' e' la seguente

$$T(n) \in \Theta(n \log n)$$

Di Lucchio Matteo - 795310 - matteo.dilucchio@edu.unit.it

Algoritmi a confronto



fonte

Consideriamo i due algoritmi Insertion Sort iterativo e Merge sort.
Dall'immagine e' facile capire come il quadrato rispetto al numero di elementi faccia crescere velocemente Insertion Sort rispetto a Merge Sort. All'aumentare dell'input il tempo di esecuzione aumenta in modo esponenziale.
E' proprio questo che mi ha spinto a provare un approccio a partizione bilanciate per entrambi gli algoritmi.

Nonostante il tentativo, l'input e' risultato troppo grande lo stesso.

Esercizio 1 - parte 2

La seconda parte dell'esercizio chiedeva di leggere i 100 interi contenuti nel file *sums.txt*, salvarli in un array e, per ogni intero, trovare una possibile somma fatta con due entry diverse dei numeri analizzati nell'esercizio precedente.

SumsFinder

Per realizzare questo esercizio ho deciso di realizzare una classe di supporto: `Sum` e di usare la struttura `HashTable` disponibile in `Java`.

Di Lucchio Matteo - 795310 - matteo.dilucchio@edu.unit.it

La classe Sum

Questo oggetto serve semplicemente a rappresentare una somma e a controllare che sia corretta.

L'uso di questa classe mi ha permesso di salvare le somme trovate in una struttura dati di tipo `ArrayList<Sum>` facilmente manipolabile.

Uso di una HashTable

L'algoritmo di ricerca in questa situazione ha una complessita' maggiore rispetto alla situazione precedente: prima bastava effettuare una ricerca per un singolo indice; ora ne servono due e l'array in cui cercare non e' ordinato.

Per risolvere a questo inconveniente mi sono servito della `HashTable`.
Ogni entry `<chiave, valore>` e' cosi' formata:

- **chiave**: il numero letto;
- **valore**: l'indice in cui e' stato letto dall'array di input.

Per trovare una somma ho usato questo algoritmo:

- per ogni elemento dell'array di somme scorri l'array per intero
- per ogni valore letto dall'array (il primo addendo)
    - se esiste un secondo addendo si trova in `hashmap(somma-chiave)`

In pratica:

Diciamo

$$F(k) = v$$

la funzione di hash che restituisce il valore v data la chiave k.
Data una somma S tale che

$$S = x + y$$

Per ogni chiave x letta dalla hashmap si ha che:
se $F(S - x)$ esiste allora esiste $F(y)$ perche' $S = x + y$ e quindi $S - x = y$.

Il problema dei doppioni

Rimane pero' un problema: ogni doppione viene eliminato perche' ogni successivo inserimento di uno stesso numero nella hashtable sovrascrive il valore calcolato dalla funzione di hash. Per risolvere a questo inconveniente ho utilizzato una seconda hashtable in cui ho salvato i numeri trovati piu' di una volta. Se un numero e' la meta' esatta della somma considerata e nell'array ce n'e' piu' di uno in questo modo e' possibile considerarlo nelle somme trovate.

Di Lucchio Matteo - 795310 - matteo.dilucchio@edu.unit.it

Complessita' dell'algoritmo

Creazione della hashtable

Per creare la hashtable viene letto ogni elemento dell'array una sola volta: $O(n)$

Ricerca del primo addendo

Si scorre l'array delle somme: $O(m)$
Per ogni somma si scorre l'array degli interi: $O(n)$
Per ogni intero si esegue una ricerca puntuale nella hashtable: $O(1)$.

La complessita' totale e' quindi:

$$O(m) + O(n)$$

Fallimento

Nonostante i test fossero positivi e i tempi di esecuzione fossero incredibilmente veloci grazie all'impiego delle hashmap, l'algoritmo non e' stato in grado di terminare a causa dell'impiego di troppa memoria per la JVM. Sperimentalmente ho potuto riscontrare che fino a 15 milioni di numeri in input consentono all'algoritmo di terminare positivamente in pochi minuti, ma con 20 milioni di numeri in input la JVM termina la memoria a disposizione. Auentando la memoria riservata alla JVM e' possibile terminare positivamente e molto rapidamente questa parte dell'esercizio.

Di Lucchio Matteo - 795310 - matteo.dilucchio@edu.unit.it

Esercizio 2 - Parte 1

L'esercizio 2 richiede l'implementazione di una libreria in grado di stabilire la distanza minima di edit per trasformare una parola in un'altra, usando le sole operazioni:

- inserimento
- cancellazione

La prima versione e' da fare ricorsiva, la seconda deve utilizzare la programmazione dinamica per dimostrarne l'efficienza rispetto alla sola ricorsione.

Entrambe le versioni si basano su 3 funzioni principali che calcolano il costo dell'operazione che rappresentano:

- inserisci un carattere: `insEditDistance`
- cancella un carattere: `cancEditDistance`
- ignora il carattere corrente e passa al successivo: `noopEditDistance`

Costi

Inserimento e cancellazione costano 1 + i costi dei sottoproblemi successivi, mentre ignorare il carattere costa 0 + i costi dei sottoproblemi successivi, ma solo se i caratteri considerati sono uguali, altrimenti ignorare il carattere ha costo infinito in quanto non e' un'operazione fattibile.

`EditDistance` ricorsivo

Questa versione calcola ricorsivamente il costo di edit senza sfruttare strutture dati in cui salvare gli esiti delle iterazioni. La stessa funzione con lo stesso input puo' essere richiamata numerose volte sprecando CPU inutilmente.

`EditDistance` con la programmazione dinamica

Questa versione calcola ricorsivamente il costo di edit tra due stringhe ma usa delle strutture dati di supporto in cui salvare i costi delle iterazioni mano a mano che vengono effettuate, in modo da evitare di reiterare la stessa funzione con lo stesso input piu' di una volta.

Vengono utilizzare delle tabelle coi costi minimi dell'iterazione alla posizione `[i][j]`, una per ogni operazione (inserimento, cancellazione, passaggio al carattere successivo) grandi `m` x `n`, con `m` e `n` pari alla lunghezza delle due stringhe - 1.

La posizione `[i][j]` rappresenta l'operazione effettuata quando `s1` e' lunga `i+1` e `s2` e' lunga `j+1`. Prima di eseguire la funzione viene controllato che il suo valore non sia gia' stato scritto in tabella: se esiste viene letto,altrimenti viene calcolato e successivamente salvato.

Di Lucchio Matteo - 795310 - matteo.dilucchio@edu.unit.it

Esercizio 2 - Parte 2

Viene chiesto di correggere ogni parola del file "correctme.txt" con le parole che hanno edit distance minino nel file "dictionary.txt".

Testando questo algoritmo con l'algoritmo ricorsivo il tempo richiesto per correggere la prima parola e' circa di 4 minuti, ma usando la versione che sfrutta la programmazione dinamica viene impiegata la meta' del tempo per correggere ogni parola della citazione e terminare correttamente.

Complessita'

- Versione ricorsiva: $O(m \times n \log n)$
- Versione che usa la programmazione dinamica: e' inferiore perche' spesso serve solo leggere il dato salvato in tabella con complessita' $O(1)$.

Di Lucchio Matteo - 795310 - matteo.dilucchio@edu.unit.it