Relazione Algoritmi e Strutture Dati 2019/20

Vannella Alessio

Indice

Sommario

Questa breve relazione spiega le scelte progettuali attuate per la realizzazione del progetto didattico del corso di Algoritmi e Strutture dati dell'università di Torino nell'anno didattico 2019/2020.

Note sui tempi misurati e sulla macchina utilizzata

Le misurazioni sono fatte su una macchina con le seguenti caratteristiche:

- Processore: Intel Core i5-1035G1 1.00GHz;
- Memoria principale : 24 GB;
- Sistema operativo: Parrot 4.8;

I tempi nei programmi implementati in C sono calcolati attraverso la funzione clock() della libreria "time.h", che restituisce il tempo impiegato dalla cpu ad eseguire il processo. Per questo motivo, i tempi di attesa effettivi potrebbero essere diversi rispetto a quelli calcolati tramite questa funzione.

I tempi nei programmi implementati in Java, invece, sono calcolati attraverso il metodo currentTimeMillis del package Java.lang.System che restituisce il timestamp del sistema al momento della chiamata. Quindi non si tratta del tempo di esecuzione del processo in cpu, ma del tempo effettivo atteso dall'utente.

I tempi sono tutti misurati in secondi.

Esercizio 1 - Analisi di algoritmi di ordinamento

È richiesta la realizzazione di una libreria che implementi due algoritmi di ordinamento: il Quicksort e l'Insertion sort. I due algoritmi devono essere in grado di lavorare su tipi di dati generici non specificati a tempo di compilazione della libreria.

Implementazione

Per agevolare l'implementazione, la libreria definisce e si appoggia ad un tipo di dato opaco *myArray*, che fornisce un set di funzioni che facilitano le operazioni su puntatori a void.
Gli algoritmi di ordinamento vengono applicati ad un file csv di 20 milioni di record con 4 campi:

- id: (tipo intero) identificatore univoco del record su cui non applicheremo algoritmi di ordinamento;
- field1: tipo intero;
- field2: tipo floating point;
- field3: tipo stringa. Si può assumere che i valori non contengano spazi o virgole;

Il pivot del quicksort è impostato sull'elemento medio dell'array.
Inoltre, il quicksort è implementato nella variante 3-Way quicksort. Questa scelta è fatta per gestire l'elevato numero di valori duplicati nel dataset di prova nel campo stringa.

Dati raccolti

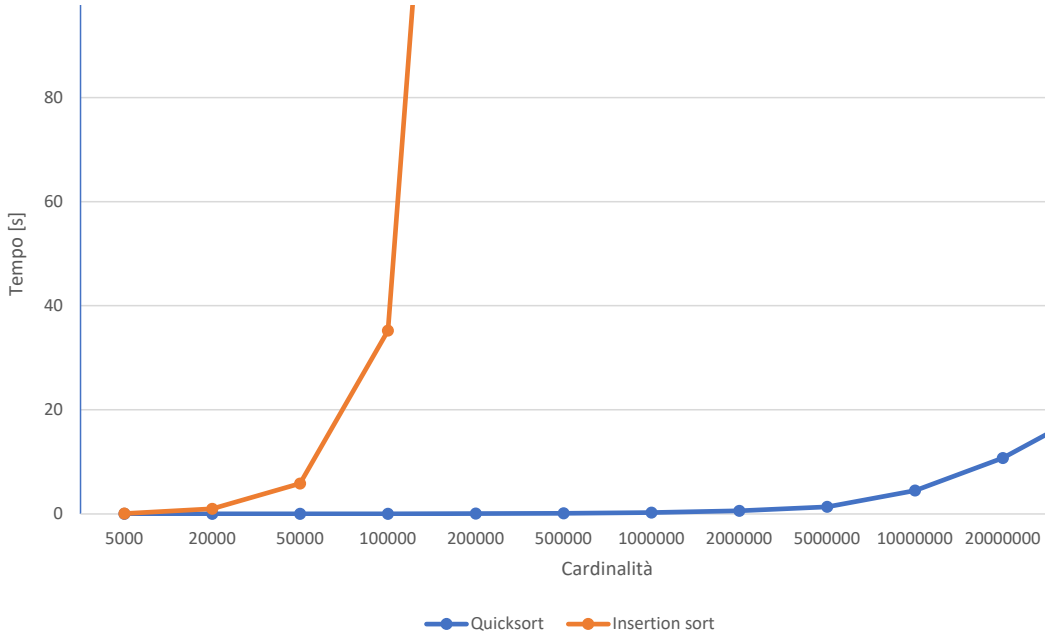
Il tempo di caricamento dei dati in memoria registrato in tabella è la media aritmetica di tutti i test fatti su quel numero di record.

		Insertion sort			**Quicksort**		
elements	**load time**	**t(s,f_1)**	**t(s,f_2)**	**t(s,f_3)**	**t(s,f_1)**	**t(s,f_2)**	**t(s,f_3)**
5000	0.005	0.060	0.066	0.121	0.002	0.001	0.001
20000	0.018	0.987	0.956	1.711	0.010	0.016	0.010
50000	0.028	5.852	7.627	14.180	0.024	0.029	0.033
100000	0.047	35.211	40.676	74.151	0.043	0.068	0.053
200000	0.093	258.125	282.824	418.370	0.096	0.116	0.159
500000	0.197	—	—	—	0.270	0.358	0.341
1000000	0.398	—	—	—	0.612	0.715	0.615
2000000	0.762	—	—	—	1.345	1.953	1.246
5000000	1.880	—	—	—	4.462	4.699	3.182
10000000	4.784	—	—	—	10.717	10.581	6.324
20000000	9.030	—	—	—	19.868	21.056	12.931

Nella tabella, f_1, f_1 ed f_3 si riferiscono rispettivamente al campo 1, 2 e 3 (interi, float e string).
Applicando l'insertion sort su un numero superiore di 500000 record (per quanto riguarda i campi interi e float) e 200000 record (per quanto riguarda le stringhe) si ha che il tempo impiegato è superiore al tempo limite di 10 minuti. Per questo motivo si è deciso di abbandonare le misurazioni sui dati successivi.

Analisi e considerazioni finali

Dall'analisi dei dati raccolti in maniera sperimentale si intuisce quello che suggerisce la teoria: il quicksort è molto più veloce dell'insertion sort, come si può notare anche dal grafico qui sotto. Nonostante entrambi gli algoritmi abbiano tempo di esecuzione nel caso peggiore temporalmente uguale a $\Theta(n^2)$, il quicksort ha complessità $\Theta(nlogn)$ nel caso medio. I risultati ottenuti sono concordi con quelli aspettati.
Inoltre, il quicksort ha tempo di esecuzione migliore sul campo stringhe, essendo l'algoritmo ottimizzato per tale caso come detto in precedenza.



Andamento degli algoritmi di ordinamento in funzione al numero di elementi processati

Esercizio 2 - Edit distance

È richiesta l'implementazione di un algoritmo che calcoli la distanza tra due stringhe s1 ed s2 prese in input. In particolare viene richiesta la realizzazione di due algoritmi. Entrambi gli algoritmi si basano su queste osservazioni:

- se $|s1| = 0$, allora edit_distance$(s1, s2) = |s2|$;
- se $|s2| = 0$, allora edit_distance$(s1, s2) = |s1|$;
- altrimenti, siano:
  - d_{no-op} : $\begin{cases} \text{edit\_distance}(rest(s1), rest(s2)) & \text{se } s_1[0] = s_2[0] \\ \infty & \text{altrimenti} \end{cases}$
  - d_{canc}: 1 + edit_distance$(s1, rest(s2))$
  - d_{ins}: 1 + edit_distance$(rest(s1), s2)$

Si ha: edit_distance$(s1, s2) = min\{d_{no-op}, d_{canc}, d_{ins}\}$

Edit distance naive

Il primo algoritmo viene implementato ricorsivamente e segue in pieno le osservazioni descritte sopra. L'Implementazione in java è autoesplicativa.
Questa tipo di implementazione è ingenua; calcola volta per volta l'edit distance di ogni sottostringa e questo la rende molto costoso dal punta di vista temporale: il limite superiore di questa soluzione è $O(k^n)$.

Edit distance con Dynamic Programming - Memoization

Proprio per la natura dell'algoritmo precedente, si può pensare ad un miglioramento attraverso l'uso della programmazione dinamica, che permette in caso siano presenti sottoproblemi ricorrenti, di velocizzare significativamente l'algoritmo. In particolare il secondo algoritmo utilizza la tecnica di memoization.
Infatti dopo un'analisi del grafo di esecuzione dell'algoritmo precedente, si può notare che ci sono molti valori di edit distance calcolati più volte.
Il secondo algoritmo sfruta proprio questo fatto e ogni volta che calcola salva, volta per volta, l'edit distance della sottosequenza calcolata in una matrice di dimensione $n \cdot m$ (con n ed m lunghezza delle due stringhe) e ritorna l'edit distance presente nell'ultima cella della matrice. Questa soluzione è asintoticamente ottima e ha tempo $O(n \cdot m)$.

	' '	**v**	**vi**	**vin**	**vino**
' '	0	1	2	3	4
v	1	0	1	2	3
vi	2	1	0	1	2
vin	3	2	1	0	1
vina	4	3	2	1	2
vinai	5	4	3	2	3
vinaio	6	5	4	3	**2**

Matrice di edit distance calcolata con DP

Considerazioni finali

Il tempo di esecuzione della seconda versione dell'algoritmo è notevolmente più veloce rispetto alla prima versione.
Dall'esecuzione dell'applicazione, abbiamo che il tempo di calcolo dell'edit distance per il dataset è di circa 23 secondi.
Un'eventuale esecuzione sullo stesso dataset con la versione senza memoization dell'algoritmo richiederebbe diversi ordini di tempo in più.

Esercizio 3 - Confronto tra Hash table e array statico

In questo esercizio viene richiesta l'implementazione della struttura dati hashmap in modo che possa accettare tipi di dati generici.
Inoltre è richiesta la misurazione dei tempi di caricamento di un dataset di prova nella struttura dati definita e in un array statico e la misurazione del get di 10.000.000 chiavi generate casualmente da entrambe le strutture dati.

Implementazione

La struttura dati implementata è un hashmap nella quale le collisioni sono gestite tramite concatenazione. In particolare, è definito un tipo di dato Node che permette di implementare una lista concatenata per ogni indice della tabella. La funzione di hash utilizzata è un semplice modulo sul numero di elementi della tabella.

Dati raccolti

L'array è un array statico di 6.321.078 elementi. Viene riordinato utilizzando l'algoritmo di quicksort che ha complessità temporale $\Theta(nlogn)$ nel caso medio, e i dati vengono recuperati attraverso l'algoritmo di ricerca binaria, sfruttando l'ordinamento dell'array.

L'hash table ha una tabella di 6.321.078 elementi. Utilizza una funzione di hash modulo limitata dalla dimensione della tabella.
Gli elementi sono generati con un seme statico basato su un contatore, in modo da rendere la distribuzione delle chiavi e dei valori casuali, ma uguali ad ogni esecuzione.

	Elementi presenti	**Elementi raccolti**	**T caricamento**	**T recupero**
Array ordinato	**6321078**	**6321929**	**1.852**	**4.950**
Hash table		**6321929**	**1.910**	**1.406**