ALBERI DI FENWICK

STEFANO MAGGIOLO

INDICE

1 Plaće

1.1 Il problema. Il primo problema di cui voglio parlare è Plaće, problema 5, gara 3, COCI 2011/2012. Abbiamo un albero, ogni nodo ha un valore associato e dobbiamo supportare due tipi di operazioni:

- aggiornamento: tutti i valori di un sottoalbero vengono incrementati o decrementati di una certa quantità;
- domanda: dobbiamo restituire il valore associato a un nodo.

Nota: un sottoalbero è identificato da un indice di nodo, ed è composto da tutti i discendenti di quel nodo (non contiene quindi il nodo stesso).

1.2 Soluzioni naïf. Come risolvere questo problema? Ci sono un paio di soluzioni abbastanza facili da trovare.

Prima di proseguire, prova a pensarci!

(1) Simulazione pura: per l'aggiornamento, andiamo a modificare direttamente i valori dei nodi del sottoalbero; per la domanda diamo semplicemente il valore.
(2) Lazy update: per l'aggiornamento, scriviamo la modifica solo sulla radice del sottoalbero; per la domanda, risaliamo l'albero e ogni volta che troviamo un nodo con un aggiornamento, lo distribuiamo sui figli (quindi risulterà completamente distribuito sul percorso che collega la radice con il nodo chiesto).

Quanto tempo impiegano queste soluzioni?

Algoritmo	Aggiornamento	Domanda
Simulazione pura	$O(N)$	$O(1)$
Lazy update	$O(1)$	$O(N)$

Infatti, anche se nel secondo algoritmo paghiamo solo la visita dal nodo chiesto alla radice, niente vieta all'albero di essere composto solo da questo percorso.

Le due strategie (sebbene la seconda sia decisamente più raffinata) hanno in comune una cosa: la disparità di prestazioni tra le due operazioni.

1.3 **Problemi con due tipi di query.** Questa situazione è tipica dei problemi che richiedono di supportare più tipi di operazioni (anche se, per essere sinceri, questi problemi non si vedono troppo spesso). Poter effettuare un'operazione in tempo costante rende difficile effettuare l'altra in tempo sub-lineare. Plaće aveva però una semplificazione non indifferente (probabilmente dovuta alla mancanza di supporto da parte dell'interfaccia), cioè le query erano date nell'input e non in streaming. Questo significa che possiamo esaminare che tipo di query ci verranno fatte e decidere di attuare una strategia piuttosto che l'altra.

Per esempio, se la maggior parte delle query sono aggiornamenti, ha molto più senso usare il lazy update, ma il fatto che sia più intelligente non lo salva dall'essere battuto dalla simulazione pura nel caso di pochi aggiornamenti e molte domande. Un'altra cosa che si può considerare, avendo la lista delle query, è di tagliare rami dell'albero che non vengono coinvolti da domande, diminuendo così la grandezza del problema.

Più frequentemente accade che le query vengano fornite in streaming da una libreria, in modo che si possa accedere alla successiva solo dopo aver dato la risposta precedente. In questo caso, non si può fare un'analisi a priori per scegliere la strategia con più probabilità di successo, e le migliori possibilità si ottengono con algoritmi che abbiano la stessa complessità per le due operazioni.

Per una soluzione ottimale (come ci si aspetta, di complessità $\log N$ per entrambe le operazioni), vi rimando alla spiegazione di Luca Wehrstedt, dato che non è lo scopo principale di queste note. Includo solo brevemente una possibile soluzione sub-ottimale in quanto l'idea potrebbe essere riutilizzabile.

1.4 **Una possibile soluzione sub-ottimale.** Il nostro scopo è sempre quello di partire da una delle due soluzioni naïf e provare a spalmare il tempo impiegato dell'operazione lenta durante l'operazione veloce, ma non così tanto come abbiamo fatto passando dalla simulazione pura al lazy update (cioè, non così tanto da invertire le complessità).

Per passare dalla prima alla seconda soluzione, abbiamo deciso che, quando ci viene chiesto un aggiornamento, non andiamo a modificare tutti i nodi del sottoalbero, ma solo un nodo. In effetti questo è abbastanza drastico, e giustifica il cambio repentino di complessità. Possiamo fare un ragionamento simile per armonizzare le complessità delle due operazioni?

Prima di proseguire, prova a pensarci!

La risposta è sì: l'idea è che quando riceviamo un aggiornamento, non andiamo a aggiornare né solo un nodo, né tutti i nodi del sottoalbero, ma una quantità di nodi pari a $f(M)$ (dove M è il numero di nodi nel sottoalbero), con $f(M)$ strettamente compreso (asintoticamente) tra le funzioni 1 e M. Chiamiamo questi nodi speciali *nodi hub*.

Quando dobbiamo aggiornare andiamo a segnare l'aggiornamento su questi $f(M)$ nodi, ma anche sulla radice del sottoalbero (che potrebbe non essere un

nodo hub). Per la domanda dobbiamo sommare gli aggiornamenti che troviamo in due tipi di nodi: i nodi hub antenati del nodo chiesto, e i nodi non-hub che stanno sopra il nodo chiesto e sotto il primo antenato hub.

La complessità dell'aggiornamento è quindi $O(f(N))$, mentre della richiesta è $O(f(N) + N/f(N))$, sommando i tempi dovuti al controllare i due tipi di nodi. È facile capire che per avere l'uguaglianza tra le due complessità bisogna avere $N/f(N) \sim f(N)$, cioè $f(N) \sim \sqrt{N}$.

Un'ultima osservazione: come decidiamo quali sono i migliori nodi hub da scegliere? Ovviamente non vogliamo che siano tutti concentrati in un sottoalbero, ma che siano distribuiti uniformemente. Quindi la cosa più semplice è efficace è di dichiarare un nodo hub con probabilità $1/\sqrt{N}$.

Il nostro scopo è quello di arrivare a parlare di un altro famoso problema con due tipi di operazioni, ma per farlo dobbiamo prima fare una deviazione.

2 Sort

2.1 Il problema. Nel problema 5, gara 1, COCI 2011/2012, abbiamo un array contenente i numeri interi da 1 a N, e lo vogliamo ordinare usando questo algoritmo: fino a che l'array non è ordinato, facciamo una passata; una passata significa partizionare l'array in sequenze decrescenti (consecutive, massimali, di almeno due elementi) e rovesciarle. Il compito è, dato l'array iniziale, dire quante volte viene chiamata la funzione di rovesciamento, sapendo che all'inizio tutte le sequenze hanno lunghezza pari.

2.2 Un'importante osservazione. Si nota subito leggendo il problema che c'è un'ipotesi molto innaturale. Di conseguenza andiamo subito a lavorarci sopra per capire qual è il motivo per cui è stata messa. L'ipotesi è ovviamente quella sulle lunghezze pari delle sequenze decrescenti (nota: in particolare implica che non ci sono nemmeno sequenze di lunghezza 1). Cosa ci dice questa ipotesi?

Prima di proseguire, prova a pensarci!

Proviamo a fare una passata. Di sicuro, le parti interne delle sequenze che abbiamo rovesciato sono crescenti, e le uniche sequenze decrescenti (non banali, cioè di lunghezza almeno 2) possono essere attorno agli estremi delle sequenze che abbiamo già rovesciato. Più importante, non possono essere di lunghezza maggiore di 2. Facendo la seconda passata, questa proprietà si mantiene: non ci saranno mai più sequenze da rovesciare lunghe più di 2 elementi.

Quindi possiamo simulare una passata e poi risolvere il nuovo problema: quanti scambi (tra elementi consecutivi) sono necessari per ordinare l'array?

2.3 La soluzione... quasi. La risposta immediata è la seguente: per portare l'elemento 1 dalla sua posizione p_1 alla prima ho bisogno di $p_1 - 1$ scambi, e così via, quindi la risposta è $1/2 \sum_i (p_i - i)$ (metà perché con uno scambio sposto due elementi). Come spesso accade, la risposta è semplice, immediata, e sbagliata. Infatti potrebbe accadere che per portare 1 da p_1 a 1, sposto in avanti il 2, cosa che magari non era necessaria.

```
for (i = N; i > 0; i--) {
    k = struttura_dati.numero_elementi_settati_minori_di(a[i]);
    if ((i + k) > a[i])
      soluzione += (i + k) - a[i];
    struttura_dati.setta(a[i]);
}
```

LISTATO 1. Lo scheletro dell'algoritmo per sort.

Possiamo però immaginare di mettere nella posizione giusta l'elemento 1, poi il 2 e così via. Come abbiamo detto, per sistemare 1 sono necessari $p_1 - 1$ scambi, e dopo non verrà più toccato. Per 2 abbiamo due casi:

- se $p_1 < p_2$, sistemando 1 non ho toccato 2, quindi per sistemare 2 sono necessari altri $p_2 - 2$ scambi;
- se però $p_1 > p_2$, sistemando 1 ho spostato 2 in $p_2 + 1$, quindi sono necessari $(p_2 + 1) - 2$ scambi.

Se proseguiamo in questo modo, cosa otteniamo?

Prima di proseguire, prova a pensarci!

Proseguendo con 3, scopriamo che per sistemarlo sono necessari $(p_3 + k_3) - 3$ scambi, dove k_3 è il numero di elementi $i < 3$ con $p_i > p_3$ (cioè, il numero di elementi in $\{1, 2\}$ che inizialmente sono posizionati oltre 3). In simboli:

$$k_n = \#\{i < n \mid p_i > p_n\}\,.$$

Fantastico! Basta quindi fare un ciclo per sommare l'i-esimo contributo, e dentro di questo un altro ciclo per trovare k_i... ma così siamo già in $O(N^2)$, e questo non ci piace. E per risolvere questo problema finalmente arriviamo all'argomento di questa dispensa.

2.4 La struttura dati. La nostra idea è quella di avere una struttura dati che ci permetta di fare l'algoritmo del listato 1, dove k è il numero di elementi che sono minori di `a[i]` e stanno a destra della i-esima posizione.

Quindi la nostra struttura dati deve gestire un array di N valori booleani (cioè in $\{0, 1\}$) e supportare la modifica di un valore e la restituzione della somma di un certo intervallo.

Se avete letto la soluzione ottima per Plaće, conoscete già una struttura dati che vi permette queste operazioni e anche altre in tempo logaritmico: un albero binario completo dove il nodo radice contiene la somma di tutto l'array, i suoi figli rispettivamente la somma della metà sinistra e della destra, e così via. Quando bisogna settare un elemento, aggiungiamo uno a tutti i suoi antenati nell'albero, e quando vogliamo la somma di un intervallo scendiamo ricorsivamente il necessario, ottenendo complessità logaritmiche per entrambe le operazioni. Sicuramente questo è un ottimo approccio e funziona, ma in alcuni casi può essere superato da un'altra struttura dati.

i	Binario	Lung. seq.	Intervallo	i	Binario	Lung. seq.	Intervallo
0	0000	0	0-0	8	1000	0	8- 8
1	0001	1	0-1	9	1001	1	8- 9
2	0010	0	2-2	10	1010	0	10-10
3	0011	2	0-3	11	1011	2	8-11
4	0100	0	4-4	12	1100	0	12-12
5	0101	1	4-5	13	1101	1	12-13
6	0110	0	6-6	14	1110	0	14-14
7	0111	3	0-7	15	1111	4	0-15

TABELLA 1. Intervalli di competenza per un albero di Fenwick con 16 elementi.

3 Alberi di Fenwick

3.1 Caratteristiche. Un albero di Fenwick gestisce un array di N interi e supporta queste operazioni in tempo logaritmico:

- aggiornamento: incremento/decremento di un valore;
- domanda: restituzione della somma degli elementi di un intervallo $[0, a]$.

Per avere la domanda su intervalli generici $[a, b]$ (potenzialmente su elementi singoli) si combinano due domande sugli intervalli $[0, a-1]$ e $[0, b]$. Rispetto agli alberi binari, occupa metà della memoria (quindi si comporta lievemente meglio grazie alla maggiore località), le operazioni di traversamento sono velocissime (tutte manipolazioni di bit, come effettivamente è se si usa un albero binario implicito rispetto a uno esplicito), e a seconda dello stile personale, può essere più facile da codificare.

3.2 Funzionamento. Internamente un albero di Fenwick è composto semplicemente da un array di N elementi, che però devono essere grandi abbastanza da contenere le somme di diversi elementi dell'array originale. Infatti, ogni elemento di questo array è la somma di un sottoinsieme di elementi dell'array originale.

Più precisamente: assumiamo che $N = 2^n$ sia una potenza di due; per sapere a che intervallo si riferisce il contenuto dell'i-esimo elemento, scriviamo i come numero binario, e consideriamo la sequenza consecutiva di 1 che parte dal bit meno significativo (anche di lunghezza zero, se i è pari). Se j è il numero che si ottiene azzerando questi bit, allora la somma è degli elementi $[j, \dots, i]$.

Le due operazioni si svolgono così (potete vedere la tabella 1 per seguire):

- per la modifica della posizione i, dobbiamo modificare tutte le posizioni che si ottengono a partire da i invertendo i bit 0 uno alla volta a partire da destra; per invertire il bit 0 più a destra si può usare semplicemente `i |= (i + 1);`
- per la domanda dell'intervallo $[0, i]$, restituiamo la somma di due numeri: il contenuto della posizione i, che sarà la somma di un intervallo $[j, i]$, più l'esito della query per l'intervallo $[0, j-1]$; come abbiamo già visto, j si ottiene da i invertendo la sequenza di bit 1 a partire da destra in 0 e sottraendo 1, e un modo per farlo è `j = (i & (i + 1)) - 1`.

L'implementazione base è data nel listato 2.

```
int N;
int ft[MAXN];
void add(int i, int value) {
  for (; i < N; i |= i+1)
    ft[i] += value;
}

int get(int a, int b) {
  if (a > 0) return get(0, b) - get(0, a-1);
  int ret = 0;
  for (; b >= 0; b = (b \& (b + 1)) - 1)
    ret += ft[b];
  return ret;
}
```

LISTATO 2. Un esempio di implementazione di un albero di Fenwick.

4 Mobiles

4.1 Il problema. Finalmente, arriviamo a parlare del problema motivatore, ovvero mobiles delle IOI 2001 (Fenwick ha pubblicato i suoi alberi nel 1994). Avete una matrice quadrata di lato N, e dovete supportare tre tipi di operazioni:

- impostare tutti i valori della matrice a 0;
- aggiungere o togliere un certo valore da una casella della matrice;
- restituire la somma dei valori contenuti in un rettangolo della matrice.

Ma queste sono esattamente le operazioni supportare dagli alberi di Fenwick, solo che qui siamo nel caso bidimensionale. Fortunatamente, la struttura dati si estende facilmente a qualsiasi dimensione: la casella in posizione (x, y) contiene la somma degli elementi posti nel prodotto degli intervalli che si avrebbero nelle caselle x e y in un albero di Fenwick monodimensionale. Come esercizio potete implementare questa struttura dati per risolvere mobiles.

4.2 Perché Fenwick. Ci possiamo chiedere perché è necessario usare una struttura dati inventata solo sette anni prima, piuttosto che, per esempio, l'albero binario completo.

Il problema specifica che N non supera 1024, mentre i valori sono interi di non più di 2 byte. Solo per tenere in memoria la matrice servono 2 MB (4 MB se non ci si accorge che si possono usare gli `short int`). Per forzare l'uso degli alberi di Fenwick, chi ha ideato il problema ha pensato bene di limitare la memoria a 5 MB, probabilmente il limite più basso nella storia delle olimpiadi.

Infatti, per usare gli alberi di Fenwick abbiamo bisogno di 4 MB, dato che ci servono tanti interi quante sono le caselle della matrice; non `short int` perché le entrate sono somme. Invece, un albero binario occuperebbe, oltre al milione di `short int` pari a 2 MB, questo numero di interi:

$$\sum_{i=0}^{10}\sum_{j=0}^{10}(2^i \cdot 2^j) - 2^{10} \cdot 2^{10} = 3141633\,,$$

dando un'occupazione totale di memoria pari a più di 14 MB.